VERIFICATION OF PLC PROGRAMS

by

Richard Šusta

Doctoral Dissertation

Submitted to the Department of Cybernetics

Revised Edition - June 2003

CTU — FACULTY OF ELECTROTECHNICAL ENGINEERING
Prague, September 2002

Submitted: September 30, 2002
Accepted: May 29, 2003

Supervisor: Prof. RNDr. Olga Štěpánková, CSc.

Opponents: Prof. RNDr. Marie Demlová, CSc.
Ing. Radek Mařík, CSc.
Dr. Ing. Radim Novotný

Address:
Faculty of Electrical Engineering
Department of Control Engineering
Technická 2
166 27 PRAGUE 6
Czech Republic

E-mail: *susta@control.felk.cvut.cz*
WEB: *http://dce.felk.cvut.cz/susta/*

The thesis was written in LaTeX 2_ε of MiKTeX distribution.

ABSTRACT

This thesis is concerned with the conversion of a PLC (Programmable Logic Controller) program into such a form, that it could be verified by some available checker tool. As we show, this conversion is a non-trivial problem.

The introductory chapter describes the motivations for this thesis and features of industrial control tasks to specify the place and properties of PLC in general.

Chapter 2 contains a short analysis of main PLCs features, summarizes some facts relevant to this thesis, and explains problems with modelling PLCs. The chapter ends by the overview of related works.

Our main contribution begins from Chapter 3 on page 31 by defining *abstract PLC* (APLC) *machine*, which is needed as universal transformation base among different PLCs. The syntax of APLC language is described and the properties of APLC machine interpreting this language are proved, mainly the termination and complexity.

APLC machine is based on binary operations with possibility of conditional calls and jumps. Its structure is very close to PLCs, which allows simple conversions of most frequent PLC instructions into APLC language.

Section 3.3 introduces into t-assignments and transfer sets, which are also novel contribution of this thesis. The necessary definitions and theorems are given and proved. This theoretical parts aim to *the fundamental proposition of this approach about the monoid of the transfer sets.* The monoid allows associative compositions of APLC operations.

In Section 3.4 on page 73, APLCTRANS algorithm based on the transfer sets is given. APLCTRANS performs very effective conversion of PLC programs into transfer sets (or logical functions respectively). The condition, under which we could expect a linear complexity of APLCTRANS, are described. The section ends by presenting experiments with 19 PLC programs, which are controlling real industrial technologies. The gained data are shown in diagrams and discussed.

The next chapter suggests several possible applications of APLCTRANS. In Section 4.1 on page 97, we show an automaton generated by PLC, whose definition utilizes transfer sets properties and gives good base for passing APLCTRANS outcome into many available checker tools.

We also prove in this section that *a PLC program can be modelled by an automaton of Mealy's family, if and only if its program is convertible into APLC language.*

Section 4.2 on page 106 deals with simplifying APLCTRANS outcome by its parallel decomposition. The soundness of the decomposition is proved and the construction algorithm is presented and demonstrated by an example.

Section 4.3 on page 118 presents the verification of races in a PLC program. The races are explained, their verification algorithm is given and

demonstrated on an example. The transfer sets can prove the absence of the races, in contrast to the algorithm published in [AFS98], which was only capable to detect some of them.

Section 4.4 on page 124 outlines composing a timed automaton with the aid of the transfer sets and APLCTRANS algorithm. An example is presented to demonstrate this conversion.

The conclusion on page 134 summarizes our contributions and contains some suggestions for a future research. The appendix overviews some elementary definitions, which previous chapters have referred to.

REMARK TO THE REVISED EDITION

Some minor errors and typos were corrected in the text. Furthermore, we have simplified some definitions in Section 3.3, so this revised edition also includes more comprehensible approach described in our later paper [Šus03].

In the revised edition, we also utilize *t-assignment* as the new name for the members of transfer sets. T-assignments have replaces former 'transfer functions' that were criticized by the opponents as a misleading notation, because transfer sets do not contain any functions.

ABSTRACT IN CZECH

Tato disertace se zabývá převodem PLC (*Programmable Logic Controller*) programů do takového formátu, aby se daly verifikovat pomocí některého již existujícího prostředku pro ověřování modelů. Jak ukážeme, tento převod není triviální.

Úvodní kapitola vysvětluje motivace, které vedly k napsání této práce, a hlavní rysy průmyslového řízení pro objasnění role a místa PLC v něm.

Kapitola 2 obsahuje krátkou analýzu hlavních rysů a vlastností PLC, ze kterých tato práce vychází, a vysvětluje problémy modelování PLC. V závěru kapitoly se uvádí přehled publikací, které se vztahují k tomuto tématu.

Naše vlastní práce začíná kapitolou 3 na straně 31, kde se definuje abstraktní PLC (APLC) interpret (*abstract PLC machine*). Ten tvoří základnu pro univerzální popis PLC a transformaci jeho zdrojového kódu. Vysvětluje se syntaxe APLC jazyka a dokazují se vlastnosti APLC interpretu, který tento jazyk vykonává. Uvádí se věty o ukončení a složitosti APLC programů.

APLC jazyk se opírá o binární operace a dovoluje podmíněné skoky a volání podprogramů. Jeho struktura je velice blízká obecnému PLC, takže nejpoužívanější PLC instrukce se do něho snadno převedou.

Podkapitola 3.3 uvádí do *t-assignments* a trans-množin (*transfer sets*), jejichž teorie je též originálním přínosem této práce. Předkládají se nezbytné definice a dokazují se potřebné věty. Tato čistě teoretická část končí důkazem hlavní věty o *existenci monoidu trans-množin, který dovoluje provádět asociativní skládání APLC operací.*

V podkapitole 3.4 na straně 73 se popisuje APLCTRANS algoritmus založený na trans-množinách. APLCTRANS provádí velice efektivní převod PLC programů na trans-množiny (nebo případně na logické funkce). Uvádějí se podmínky, za nichž můžeme očekávat lineární složitost algoritmu. Podkapitola končí přehledem experimentů s 19 PLC programy, které řídí skutečné průmyslové technologie. Uvádějí se a rozebírají se získaná data.

Další kapitola uvádí některé možnosti aplikací APLCTRANSu. V podkapitole 4.1 na straně 97 se popisuje automat generovaný PLC programem, jehož definice využívá vlastností trans-množin. Automat poskytuje dobrou základnu pro přenesení výstupu APLCTRANSu do řady dostupných nástrojů pro ověřování modelů. éto podkapitole také dokážeme, že *PLC program lze modelovat pomocí Mealyho automatu tehdy a jen tehdy, když jeho program je možné vyjádřit v APLC jazyce.*

Podkapitola 4.2 na straně 106 se zabývá zjednodušením APLCTRANS výstupu pomocí paralelní dekompozice. Dokazuje se postačující podmínka její existence a uvádí se konstrukční algoritmus tohoto automatu, který je demonstrován na příkladu.

Podkapitola 4.3 na straně 118 se zabývá zjištěním stavové nestability (*races*) v PLC programu. Vysvětlují se příčiny této nestability a uvádí se

algoritmus pro její zjištění spolu s ukázkovým příkladem. Trans-množiny dovolují vyloučit nestabilitu, na rozdíl od algoritmu, který byl publikovaný v [AFS98]. Ten nestabilitu pouze detekoval v některých případech, ale nedovedl dokázat její neexistenci.

Podkapitola 4.4 na straně 124 uvádí postup vyjádření PLC jako časového automatu (*timed automaton*) s využitím trans-množin a APLCTRANS algoritmu. Metoda se ukazuje na jednoduchém příkladu.

Závěr na straně 134 shrnuje naše příspěvky k tématu a obsahuje i některé náměty pro další výzkum. V příloze lze pak najít přehled několika základních definic, na než se odkazovalo z textu.

ACKNOWLEDGMENTS

I am most grateful to my supervisor, Prof. Olga Štěpánková, who inspired me to the verifications of PLC programs.

I would like to thank Prof. Vladimír Mařík for possibility to participate in the projects of Rockwell Automation Allen-Bradley's laboratory, where I have made my first contacts with PLCs, and to Ing. Jindřich Fuka, the leader of the laboratory.

I would like also to acknowledge utilizing experience in the thesis that I have gained by participating in several industrial projects led by Ing. Vladimír Lhotský (the chief of the control division of Blumenbecker Prague), Dr. Radim Novotný (the manager of SIDAT), and Ing Jiří Svoboda (the director of SPEL).

Finally, I am indebted to my colleagues from Department of Control Engineering and mainly to my wife, Dagmar, for encouragement and support of my work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Verification of Programs

This thesis described herein is concerned with the verification of PLC programs or, more precisely, with the conversion of PLC program into such a form, that could be verified by some available model checker.

PLC (Programmable Logical Controller) is an abbreviation for wide range of products. Numerous manufactures offer many PLC types, from simple compact models with fixed inputs and outputs to modular PLCs with a variable structure. Some PLCs are also capable of collecting data by networks and advance PLCs offer a multitask environment.

In practice, the development of PLC program roughly consists of four overlapping phases:

1. designing and writing required code,
2. initial testing of new program,
3. commissioning i.e., putting automated technology in operation, and
4. the fine-tuning of a PLC behavior.

The formal methods can help with all phases besides the commissioning, but mainly with the initial testing and the fine-tuning.

The first phase depends on analysis of control aspects of a technology and used techniques usually belong to other fields of science. Although there were done several attempts at developing tools for writing formally verified PLC code, for example MobyPLC [DFT01, TD98], this techniques are limited to a subset of control tasks.

The second phase, initial testing of written program, is nowadays provided frequently by entering selected sequences of input data and monitoring responses of PLC. This procedure cannot discover all errors. If the program did some operation several times without a serious bug, this is no proof of the absence of some fatal bugs. Such conclusion could be valid, under very precise conditions, for hardware but not for software. Hardware failure modes are generally much more limited, so their testing and building protection against them is usually easier. For example, if we need to know the reliability of a relay type we can connect hundred identical relays to a testing circuitry, million times switch them on and off and estimate their reliability.

No similar procedure is fully applicable to software. Software does not subject to random wear-out failures like hardware and it does not degrade due to wear, fatigue or reproduction process like mechanical elements, therefore nonsoftware professionals sometimes put too much confidence in its reliability. However, design errors in programs are much harder to find and eliminate because software behaves in different way than mechanical world. Million successful runs of a program do not generally prove low probability of dangerous malfunctions because all possible paths in the state diagram of a program can easily exceed the million by million times and frequently much times more.

Users express a dissatisfaction with manual rigorous testing of software since 1940s, with the development of more complicated programs. This technique received nickname 'debugging' (reputedly after a dead moth that was found inside contacts of a relay when Mark II computer was developed). Debugging is frequently used method but very poor tool for the safety diagnostic. E.W. Dijkstra's saying is well known: "*testing can reveal only the presence of errors, not their absence.*"

The testing of a PLC program could be improved by a model checker (e.g. SMV [McM97], SPIN [Hol97], UPPAAL [LPW97], KRONOS [DOTY96], and others) but this way is used rarely for some reasons like:

- The usage of a model checker requires some expertise. Known checkers are far from an ideal situation: "enter a program and propositions and read results".
- Model checkers require describing verified operations in special languages. Any manual conversion of a PLC program to a checker code can devalue results by adding new errors.
- Existing checkers are capable of analyzing models with limited complexity and even very simple PLC programs need many variables; $n \geq 100$ variables is rather a normal than an unusual case. This gives a theoretical number of PLC states is 2^n.
- A manual decomposition of a PLC program to simple blocks can be too laborious.

We consider that the primary problem of PLC verification lies in the conversion of written PLC program into a proper form. We suggest that this tool should strictly operate on the base, which could be phrased as the following. *Enter a PLC code in the tool, and obtain the output suitable for the most of checkers. Any manual adjustments of data should be reduced as much as possible.* If such tool had existed, it would open possibility for employing many verification techniques.

The formal verification of a PLC program when an introductory testing is performed, prolongs this phase. However this extra effort will facilitate and hopefully speedup the following phases. Every error that was discovered will short the commissioning. This critical and very expensive phase compares our internal model of the technology in question with the reality.

A controlled program was written with respect to some presumptions, which need not be necessarily correct or complete. For example, used indicators could give output signals with inverse polarity than planned, or wires were connected to another I/O post mistakenly. Therefore, *no formal verification can substitute practical putting PLC programs in an operation.*

When the technology runs successfully, other malfunctions are discovered gradually. PLC program is fine-tuned for month, years. Based on personal experience, not only with my own programs, but also with the other ones, I daresay that no complex PLC program is error free. Technicians have popular saying: "PLC program will be definitely debugged only when the controlled technology is already totally worn-out and scrapped."

The formal methods could help to locate very rare malfunctions, whose manual finding represent sometimes so exhausting process that workers prefer accepting them like 'necessary component'. I personally encounter this approach many times.

The funniest story happened to me in a Slovak cement factory where I noticed a hammer suspended on a thin chain beside a big industrial relay. When I asked a production foreman for the purpose of the hammer he explained to me that the relay needed sometimes a small stroke to fix its malfunction and workers hung the hammer here not to bring it with them all the time. I logically asked why the bad relay had not been replaced. The foreman answered with serious face that replacing the relay could have caused a lot of troubles. The malfunction could have moved to some worse accessible component and so it is better to let *her* stay here.

I did not laugh at his personification of malfunctions. Three good reasons kept me quiet. At first, it was potentially possible that a malfunction of the relay could disable another problem. I several times experienced appearing new bugs after fixing a previous one. At second, I understood foreman's point of view. Workers were upset by too many bugs. They had not fully gotten accustomed to newly automated machines yet and therefore they felt happy to encounter the classic relay malfunction. Finally, I had no reason for laughing because I arrived in the factory to fix my own program; filling railways wagons with cement powder sometimes stopped without reasons and the restart button had to be pressed to continue. It was not the fatal malfunction, only troublesome, because I programmed auxiliary operations allowing to go on without loosing information. Writing control program I calculated with possibility of hidden errors undiscovered by the testing.

The situation that some errors 'survived' all rigorous tests and turned into an 'integrated' part of a machine happens more frequently than it is generally taken into account. The investigation of deadly accidents in safety-critical systems involved a computerized radiation therapy machine called the Therac-25 brought also attention to the fact that operators accepted frequent low dose malfunctions as 'normal behavior'.

> ' She said [hospital physicist]: *"It was not out of the ordinary for something to stop the machine... It would often give a low dose rate in which you would turn the machine back on... I can't remember all the reasons it would stop, but there [were] a lot of them."* A radiation therapist at another clinic reported an average of 40 dose-rate malfunctions, attributed to underdoses, occurred on some days.' Quoted from [LT93] .

The operators worked on the Therac-25 machine demonstrating a lot of evident imperfections in its control software, but hospitals continue using the machine until second Tyler's overdose (fifth dead accident in the order). The manufacturer of Therac-25, Atomic Energy Commission Limited (AECL), had tested its control program for 2,700 hours of use and did not find out two mortal bugs in it [LT93]. The bugs also 'survived' detailed inspections of Therac-25 which had been performed after the previous dead accidents. Proper formal methods should locate the bugs in few hours or days.

1.2 Reactive Software

In this part, we specify the range of tasks in general, to which we aim our attention. The formal verification is too wide and difficult task. Many methods have general validity but the selection of a suitable method depends heavily on our requirements. In practice, all verification methods are limited to a part of programs and universal tools are hardly ever given.

The Harel and Pnueli [HP85] divided universe of computer programs from point of computation into those which are *transformational* and those which are *reactive*. The typical representative of a transformation program is the function $f(y_1, y_2, \ldots, y_m) = (x_1, x_2, \ldots, x_n)$ producing upon termination the final values $y_1, y_2, \ldots, y_m$ from input parameters $x_1, x_2, \ldots, x_n$.

Francez [Fra92] calls such program the state (or value) transformer and characterized it as follows:

> The 'task' of the [transformation] program is to compute, starting from some initial state, for some finite amount of time, and

then to terminate by producing some final state or outcome of the computation, having some desirable properties.

By contrast, a reactive program, the topic of this thesis, is one of that maintains an ongoing interaction with its environment and responds to dynamically changing inputs by producing stream of outputs. In such cases, speaking about a reactive program means speaking about combination of program and hardware, the both being embedded in an environment.

1.3 Process Control Environment

An important environment for verification is the process control. Every serviceable method, even that capable of finding out a small subset from all possible bugs, can prevent huge financial losses. To specify process control environment, we build the hierarchical model of a production that will be derived from general CIM model [Slo94]. Figure 1.1 depicts the model modified for a common industrial process.

Technological process level lies at the bottom grouping all machines, production lines, drives, and the like. It also includes the sensing units and actuators necessary for the automatization and tools for communications with workmen, like information indicators, displays, command elements for manual actions, and so on. The automatization components used here are mainly prefabricated and their usage requires only setting proper parameters, either manually or with the aid of supplied programs.

The control is done from *Logical control* level by programmable logic controllers (shortly PLCs), specialized computers designed for maximum reliability and harder environment conditions. We will aim our attention to them in next subhead.

At this point, we must highlight that the boundary between the two lower levels of our model is more organizational than physical. The elements included in *Technological process* can be physically placed in some PLC, or plugged in a PLC rack when a modular PLC is used. For that reason we should talk more precisely about PLC processors i.e., the central units of PLCs, which control *Technological process* in our model. Hereinafter we will call PLC processors shortly PLCs in accordance with common practice in PLC literature. If we specifically have in mind a PLC as the electronic device consisting of PLC processor and other support modules, we will always emphasize this fact.

The top level, *Supervising*, supervises whole process, checks for its safety conditions and modifies its run according requirements of planning and management. *Supervising* level includes systems for visualizations of the process flow and operator's interfaces for setting required data.

Concentrating on PLC we can simplify our previous model to the embedded model depicted in Figure 1.2. The model consists of a program, a

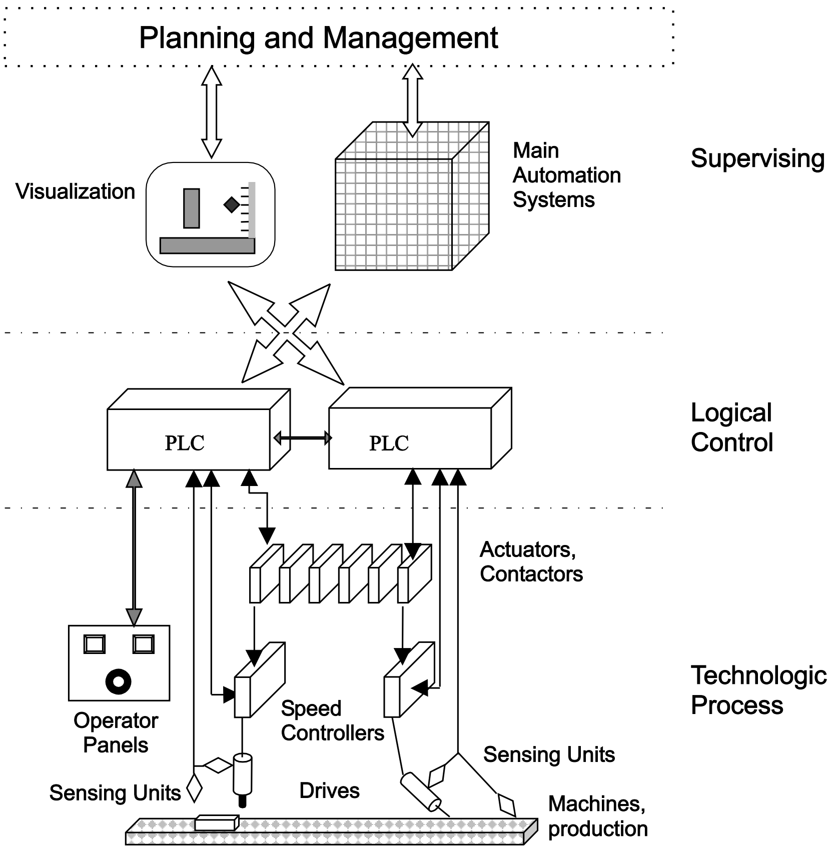


Figure 1.1: Hierarchical Model of Process Control

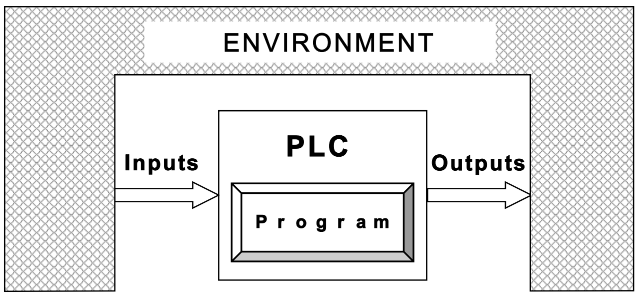


Figure 1.2: PLC as Embedded System

PLC on which the program runs, and environment in which PLC hardware runs. The environment means not only process itself to be controlled (a machine, vehicle, aircraft, or a whole factory), but also cooperating components (another PLCs, monitoring devices, or operator panels).

We can present few assumptions for the environment that the part of PLC applications could possibly satisfy. They will not hold for all of them certainly, thus the following list should be considered as *some better case for PLC verification*:

1. All analog control loops are closed in *Technological process*, for example with the aid of special external modules, as speed controllers and the like.

   This assumption excludes hybrid programs i.e., those controlling both analog and logical devices. This is acceptable, because utilizing specialized elements is a common practice for faster analog processes or dangerous peripherals as drivers, and PLC programs usually control only minor processes with slow time constants, for example heating and so on. The specification does not rule out all hybrid PLC programs. Under suitable conditions, there is possible to split verification to an independent analog and digital part.

2. PLC hardware is connected to the PLC environment by digital input and output signals, as a contact is either closed or opened, or a drive is either running or stopped. Hence inputs and outputs contain only binary or finite integer variables and their values sampled in some time will fully describe momentary state of PLC environment.

   Some PLCs utilizes peripheral devices that require overlapped data transfer. After initialization and starting the transfer is performed asynchronously with PLC program flow and its termination is signaled by an event. These devices usually need special handling algorithm that have many features of parallel programming and their verification is more difficult. [1]

3. *Technological process* takes into account unexpected switching any PLC control off. In that case mechanical and electrical arrangements of *Technological process* will automatically move all uncontrolled machines to harmless positions to prevent things fall down, and the like.

   This assumption follows from common work safety rules that demand immediate interrupting production in dangerous situations. Stopping PLC will naturally result in suspending "Technological process", either whole or its part. Consequences of such break will depend on the controlled technology. Many technological processes allow continuing without a serious lost, with exception of time. The practical consequence of the specification is resetting a part of outputs when initializing a PLC program. *Thus we have initial states!*

4. All control operation are provided on *Logical control* level.

   This describes common situation in PLC control applications. The safeties of prefabricated automatization components, utilized in "Technological process", are mainly matters of their manufactures. The proper wiring, arrangement and setting are usually only possible verifications left to automatization engineers. On the contrary to this, the verification of "Supervising" level is nearly impossible, because there are usually utilized higher level computers (as PCs or workstations) programmed by rapid development environments specialized for supervisory tasks, for example RSView (Rockwell Automation), WinCC (Siemens AG), or Operate IT (ABB). Under favorable circumstances the flow diagrams of user's programs or user's data can be formally verified, but not the computer operating system itself and runtimes library supporting applications. The stability of "Supervising" level will always be unknown factor and therefore critical control operations are programmed at *Logical control* to ensure *Technological process* independent on *Supervisory* level.

[1]Such data transfers are used only for either multichannel analog modules or highly specialized units, that operates as independent devices, for example obtaining visual information [FŠ92, Šus93].

Chapter 2

Background and Related Works

2.1 Overview of PLCs

In this section we review some of PLC characteristic for the convenience of all the people who need to acquaint more with them.

There are many books about PLCs but, in fact, most of them present technical details of a specific PLC type like configurations and the sets of instructions as manuals [Roc98], [Roc01b], or [Sie02].

Theoretical overviews of PLCs appear rarely. They usually do not cover some advanced features and concern fundamental principles needed for engineers and technicians [Hug89]. Some brief information can be also found in PLC related publications [BHŠ00], [Die00], and [Tou00]. Therefore, we have written this part, which overviews some facts relevant to next chapters.

In general, a PLC behaves more like regulator than an ordinary computer. A PLC program is executed in the cyclic manner, depicted in Figure 2.1. One general PLC cycle consists of:

- Polling inputs, or sampling respectively, and storing their values into inner memory. This phase is called *Input scan* in PLC manuals.

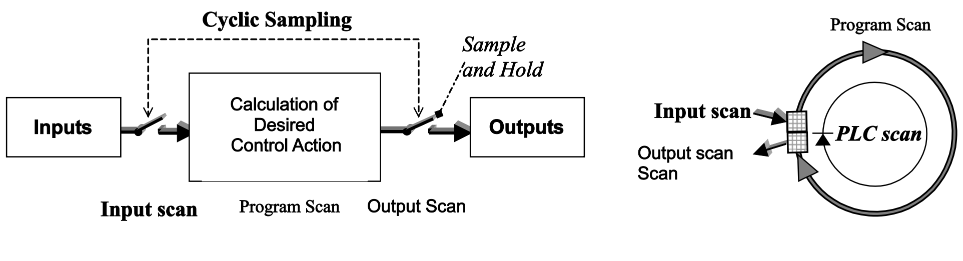


Figure 2.1: Principal Schema of Classical PLC

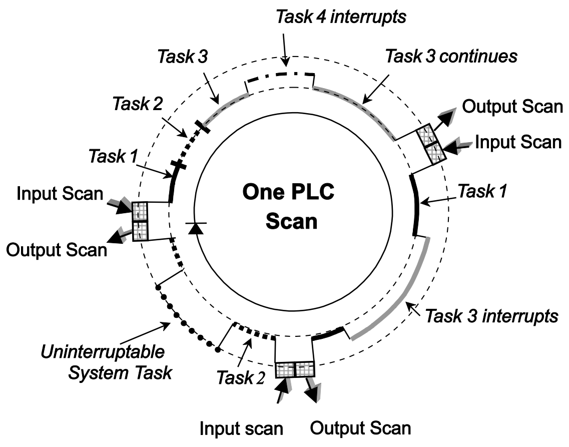


Figure 2.2: Example of a PLC Scheduling

- Calculating the proper control action. The operating system of the PLC executes the user's program once. At this point, the program makes arbitrary computations and changes the values in special memory for outputs. This phase is called *Program scan.*
- Writing outputs to peripherals. This phase is called *Output scan.*

The circle shown on the right side in Figure 2.1 symbolizes all three phases. They can be easily programmed on nearly any computer with the aid of an endless loop, with the exception of efficient polling numerous input and an output signal in hard industrial environment. This polling is the primary domain of PLCs and it mainly distinguishes them from other computers.

Older PLCs or simple types strictly operate according to Figure 2.1. Advanced models are capable of running several tasks and they have not strictly bound program scans to input and output scans, as shown in Figure 2.2. Their scheduling comes near to high level operating systems with only one exception. PLCs have no direct human graphical interfaces and therefore they need not quickly switch between processes to create illusion of fluent drawing. If no task with higher priority is waiting, each running program can be executed all at once.

We label *activation of a task* the operation of PLC scheduler that begins immediately scanning of some task if no scan with higher or equal priority is momentary done, otherwise the task is added into a queue in which tasks wait for their scan. Four main types of *activation of a task* use to be found [Roc01a]:

Continuous task. The task is held permanently active. When one its scan is finished, the second scan is automatically started at the first possible moment;

Periodic task. The task is set active in periodical time intervals;

Event-driven task. The task is set active if some event occurs.

Sequential task. The activation of the task is controlled by a special sequential program that is usually based on Grafcet [Int92]. The name and syntax depends on a manufacture, for instance, Allen-Bradley PLC have sequential program SFC (Sequential Function Chart) and Siemens PLC series S7 use S7-Graph. Sequential tasks comprise a higher level of PLC programming. They are suitable only for controlling processes, which consist of series of distinguishable technology operations, and therefore their verification is not included in this thesis. It is studied in several other publications, for example [BM00] or [Tou00].

Practically every PLC has at least one continuous task that uses to be its primary task. Most of PLC types (even simple ones) allow one or more periodic tasks. The event-driven tasks are often reserved for PLC operating system. They are activated on errors or selected situations, as *the first scan after PLC power up that serves for initializing variables*, [1] but some PLCs allow programmer utilizing event-driven tasks for monitoring changes of inputs.

The second important changeover against Figure 2.1 in advanced PLCs concerns input and output scan. PLCs utilize distributed data collections by specialized modules connected throughout control area networks, as depicted in Figure 2.3. Such solutions minimize the total length of necessary wires (usually the faultiest components of an automatization) from PLC to controlled elements.

Using networks adds delay to the reading and the writing of external data. If all input and output data are polled in one-stroke, scan times increase inadequately. For that reason, some PLCs are designed with I/O channels operating independently to the program execution. Their polling is not synchronized with the program scan and reading and writing I/O data depends only on predefined refresh times for individual input or output. In such case, the PLC scan cycle, in Figure 2.1, consist of only one phase — *program scan.*

In such environment, PLC inputs and outputs are usually divided into scan groups and I/O scans are replaced by asynchronous operations, which periodically sample input values or send outputs to the peripheral devices in intervals that depend on individual settings of the scan group which data belong to. If an program operation needs inputs that do not change during whole program scan, then it must begin by copying inputs into some variables, which it will use i.e., it performs its own input scan. Similar duplication must be done for the outputs at the end of the operation. These

[1]We utilize this first scan initialization in Section 4.1.

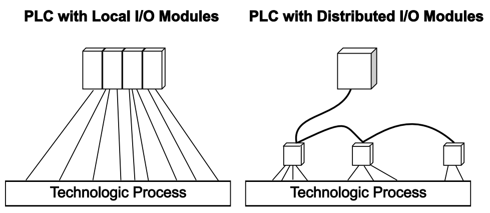


Figure 2.3: Distributed I/O Modules

copying could be also used for repairing inputs or outputs (see page 98). Therefore, if we verify a PLC program, we should take in account that:

- the inputs or outputs could be moved to different addresses in the program,
- the *inputs can be ordinary variables stored in PLC memory* i.e., some instructions can possibly write to an input, [2] and
- we must add to inputs also variables that are generated by PLC operating systems, for example "first scan" signal mentioned above.

Software - PLC Programming

In 1993, the International Electrotechnical Committee (IEC) published the IEC 1131 International Standard for PLCs [Com]. The part 3 of this standard defines a suite of programming language recommended to be used with PLC. The standard defines the semantics of these languages mostly by way of examples. It focuses attention mainly to graphical languages that substitute elder forms relay diagrams and logical function.

Ladder Diagram (*LD*) - graphical language that appears as the relay diagram schematics;

Function Block Diagram (*FBD*) - graphical language that is based on rectangular boxes interconnected by means of connection and connectors. It appears like a schematic of logical function;

Instruction List (*IL*) - language resembling a typical assembler;

Structured Text (*ST*) - textual language with PASCAL like syntax and functionality. Its meaning fades in comparison to other tools.

[2]There are also some read-only inputs in PLCs, either protected by some settings or generated by PLC system, but all attempts of changing their values are recognized by PLC programming software and announced as errors. Thus we may suppose that a program writes only to variables, which may be changed.

$$\begin{tikzcd}
\cdot \arrow[r, "A"] & \cdot \arrow[r, "\overline{B}"] \arrow[r, bend right, "C"'] & \cdot \arrow[rr] \arrow[dr, "\overline{D}"] & & Q1 \\
& & & \cdot \arrow[r] & Q2
\end{tikzcd}$$

Figure 2.4: Directed Ladder Graph for Figure 2.5

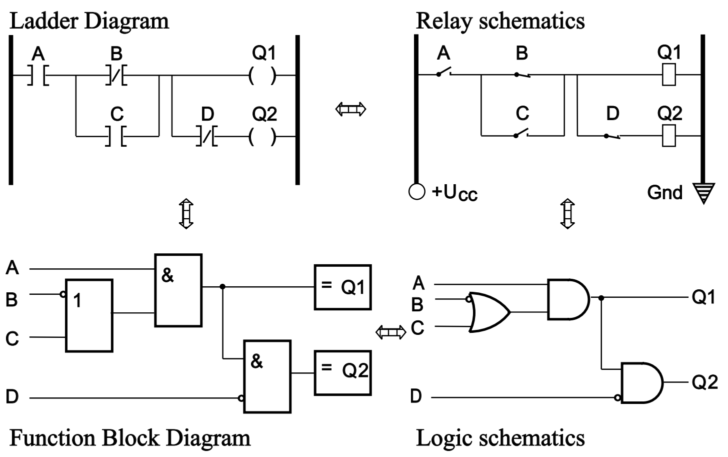


Figure 2.5: Ladder and Function Block Diagrams

Both graphical languages (ladder or logical diagram) were designed primary for easy inspection of a running program, because some technologies require continuous operation to achieve higher quality of products and malfunctions are repaired without stopping PLCs (if it is possible).

Suppose, that simple control program shown in Figure 2.5 has its output variable *Q2* connected to a drive *Q2drive* through an output *Q2out*:

$$\begin{tikzcd}
\text{Program} \arrow[r, "Q2"] & \text{Output module} \arrow[r, "Q2out"] & Q2drive
\end{tikzcd}$$

When *Q2drive* does not start as expected, a called janitor first tests the drive *Q2drive* and measures the PLC output. If none of the above reveals a cause of the error, then he will have to find out, which one of input conditions A, B, C, or D has failed. The inspection of the program, either looking into PLC program or in its listing, offers one of possible guidelines to gain such information. This method has advantage of low purchase cost — no special diagnostic software need to be added to the written control program.

The ladder diagram converts the satisfying logical functions to simple searching for a broken or unbroken path in the ladder graph that is strictly directed from left to right, as shown in Figure 2.4. Janitors can perform it

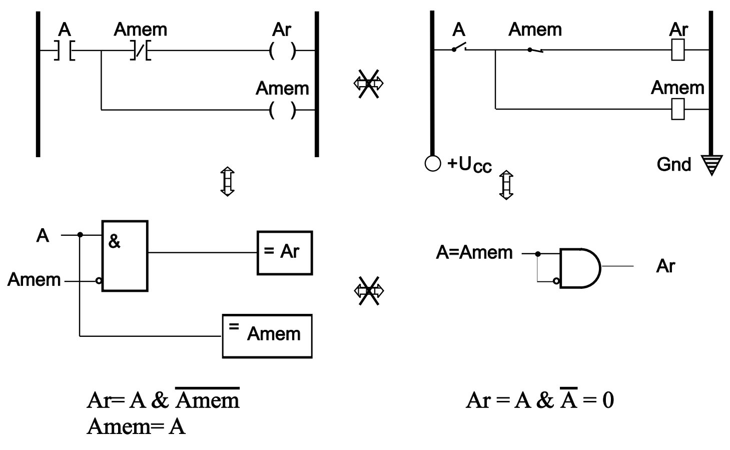

A
Amem
Ar
Amem
A
Amem
Ar
Amem
+Ucc
Gnd
A
Amem
&
= Ar
= Amem
A=Amem
Ar
Ar= A & $\overline{\text{Amem}}$
Amem= A
Ar = A & $\overline{\text{A}}$ = 0

Figure 2.6: Rising edge detection

without a deeper software training, because the task is analogous to solving errors in contact arrays.

On the other side, the similarity of graphical representations to hardware can misguide, because the both graphical languages do not emulate exactly their corresponding hardware patterns. All parts in relay or logical circuits operate parallel, in contrast to software emulations of ladder and function block diagrams that are performed in predefined order: usually from left to right, from upper to lower elements.

Simple program depicted in Figure 2.5 can be considered as the direct replacement of relay and logical schematics, but the program for the rising edge detection depicted in Figure 2.6 has no hardware mirror. The left side diagrams define famous rising edge detection: the output Ar will be set to $true$ for one program scan if A has changed its state from $false$ to $true$, othewise Ar remains in $false$ state. The corresponding hardware circuits will hold Ar output permanently in $false$ state, however Ar relay can accidently close for short time if $delay_{close}(Amem) > delay_{close}(Ar)$.

The graphical languages are usually stored in PLC processor as some list of instructions that is evaluated by PLC operating system. Thus we may consider graphical languages as some visualization of processor codes.

Instruction lists are nowadays frequently implemented as primary tool, for example in Siemens PLCs, or at least like an alternative to a graphical language, as in Allen-Bradley PLCs. The disadvantage of instruction lists, in terms of a program analysis, is their dependency on PLC type.

However, as we show in the thesis, a universal base exist that allows expressing subset of PLC instructions in a common abstract language. We will deal with this problem in the next chapter.

PLC correctness

The correctness of real-time programs not only depends on the functional aspects of their operation, but also on the timeliness of their behavior. Tourlas divides the PLC analysis [Tou00, pg. 18] in his thesis into two separated questions: *logical correctness* and *temporal correctness*. Such classification is not fully sufficient because it does not distinguish time aspects of hardware from program. A program can show proper behavior under good configuration and very poor after modifying environment. Replacing fast PLC processor by a slower type offers an example clear at first glance, but the modification need not be such significant. For example, addition of a new distributed I/O modul can possibly decrease scan times of inputs and outputs, if PLC network configuration is near its critical point [Nov01] and another node added to network results in an exponential increasing of transmission delays.

Therefore, we should distinguish three type of PLC program correctness:

logical correctness - proper functional behavior of the program that is specified by logical or temporal logical statements without explicit time constants ;

time correctness - timeliness behavior of the program that is specified by logical or temporal logical statements with explicit time constants. Time characteristics of PLC processor and its environment are neglected ;

PLC correctness - verification of the program behavior on a predefined PLC or a set of PLC types.

Lemma 2.1 Time correctness *does not generally imply* logical correctness *and vice versa.*

Proof: The proof can be done by the example. If programs satisfies *time correctness* defined by the specification "the $state_n$ will not be reached from $state_0$ in time $t \leq \tau$", then that says nothing about a validity of *logical correctness*: "$state_n$ will [or will not] be reached" and vice versa. □

Lemma 2.2 PLC correctness *does not generally imply* time correctness *and vice versa.*

Proof: If we consider simple program for powering up a lamp L after closing a switch S, then the program with single instruction $L := S$; does not satisfy *time correctness* defined by the proposition: "L will close with the delay greater than 100 miliseconds after switching S". In contrast, if PLC

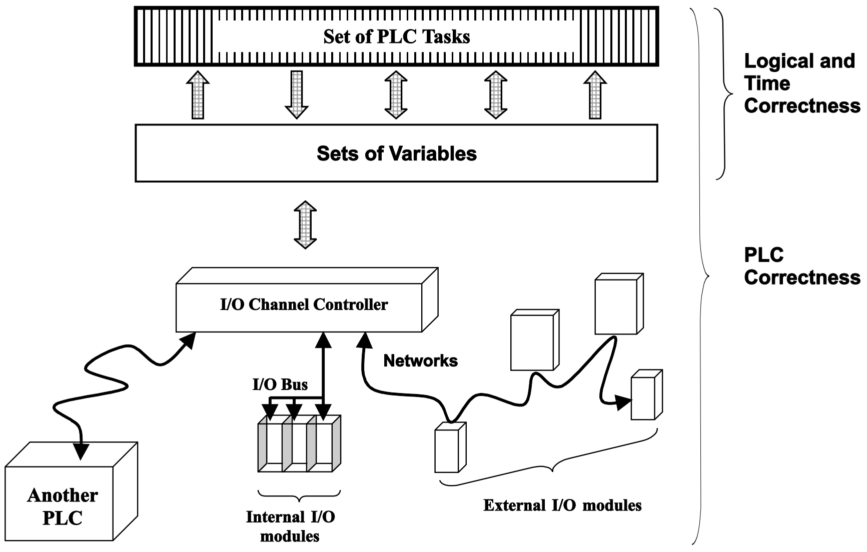


Figure 2.7: Correctness of PLC Program

correctness is considered then the proposition could be satisfied when running the program on a *very very* slow PLC. □

It follows from considerations above that each correctness forms a different question. PLC correctness is the most interesting case for practical applications, but it depends on used PLC hardware and its configuration. On the other hand, if PLC hardware is properly chosen and configured, then delays added to I/O signals will be nearly negligible.

Under favorable conditions, *time correctness* specified by a set of propositions Φ, in which a finite set of time constants $T = \{t_1, t_2, \ldots, t_n\}$ is used, could imply *PLC correctness* $\bar{\Phi}$. $\bar{\Phi}$ contains propositions similar to those in Φ, but with modified times $\bar{T} = \{\bar{t_1} \mid \bar{t_2}, \ldots, \bar{t_n}\}$. The number $e = \max_{i=1}^{n} |\bar{t_i} - t_i|\, ; \bar{t_i} \in \bar{T}, t_i \in T$ denotes the accuracy of the approximation *PLC correctness* by *time correctness.*

Definition 2.1 *If time correctness specified by a set of propositions Φ approximates PLC correctness with a required accuracy σ then PLC is called* σ-well-designed with respect to Φ.

We must emphasize the fact that time correctness could imply PLC correctness only if the propositions define 'reasonable' constraints. The adjective 'reasonable' means requirements realizable by an available hardware and for a controlled technology. Those conditions depend on many factors and relate to control engineering know-how. Therefore, exact mathematical definitions can be hardly given.

The proposition: "L will close exactly t milliseconds after switching S" represents an example clear at the first glance. A program can satisfy time correctness, but PLC correctness will not be fulfilled because the requirement is similar to comparing a float point number by equality operator.

On the other hand, PLC correctness specified by another proposition: "L will close if S is switched for a time interval long from t_1 to t_2 milliseconds" is questionable. It will depend not only on delays of used I/O units but also on their sensitivity to temperature and other working conditions. If the delays are too unstable, then no assumption about L lenght can be given.

To gain results with general validity, we will suppose *σ-well-designed PLC* and exclude PLC correctness from further studies. PLC programs will be observed as tasks executed in some variable environment, as shown in Figure 2.7.

Modelling PLC

Good overview of PLC models was written by Mader [Mad00a] who classifies the models according to three orthogonal criteria:

- to what extent and how PLC processor operation is modelled;
- the use of PLC timers in instruction set;
- the fragment of programming languages that is possible to analyze.

The first criterium deals with PLC cyclic behavior. Scan cycle times depend on many complex circumstances, which give them random behavior. In general, scan times can be expressed better like probability statements than implicit equations. Therefore, PLC models include time characteristic of PLC scan by the four ways:

Static PLC models do not consider scan time characteristics and they are intended only for static analyses as in [BHLL00], where was used abstract interpretation operating with sets of intervals, in which we could expect the value of a variable.

Abstract PLC models describe cyclic behavior of PLC program but they simulate all phases as operations taking place in zero time;

Implicit PLC models will follow from the abstract models if we add assumption that the time of scan cycle is a constant i.e., each phase of PLC scan takes always the same exact time;

Explicit PLC models can be obtained from the implicit models by adding the constraint that the each scan cycle is forced to take time laying between the lower and upper time bound. The model respects variable durations of PLC scan as a probability distribution, example can be found in [Die00].

Although the sequence above can be continued to models with more precise PLC scan characterization, such extension have probably no practical significance. Scan times of real PLC depend on many complex circumstances and the time constants needed for implicit and explicit PLC models are usually either predefined input constrains or results obtained by practical experiments with PLC programs.

Moreover special solutions for time critical regulations are offered by modern PLCs, as high priority periodical tasks scanned in short regular intervals. Therefore, detailed analyses of scan times need not be primary problem of the PLC verification.

When designing a PLC model more serious question is the presence of *timers* in a recognized instruction set. If timers are not allowed the verification is limited only to simple programs what is unsatisfactory. Practically every real PLC program contains timers because the duration of many output and input signals must be controlled. *Timers substitute for incomplete knowledge of the PLC environment.*

For example, suppose that we control the transport of paper boxes on a conveyer that has two sensors, at its begining and end. We know that every box should leave the conveyer at most 10 second after passing the beginning sensor. Watching for a box jam means either adding extra expensive sensors along the conveyer or programming one timer that will check transport times of the boxes.

2.2 Related Works

Many publications concern a verification of PLC, but few works include some conversion method of PLC programs. We have found only three published algorithms.

PLC conversion was studied by M. Rausch and B. H. Krogh from Carnegie Mellon University [RK98], whose algorithm one algorithm for converting ladder diagrams into SMV. The author supposed the following constraints (quoted from [RK98, page 3]):

- only Boolean variables are used;
- single static assignment, that is, variables are assigned values only once in the program;
- there are no special functions or function blocks; and
- there are no jumps except subroutines.

The similar approach was chosen by Corporate Research Center of ALCATEL (Marcoussis, France) and and Ecole Normale Suprieure (Cachan, France). Their two papers [DCR^{+}00], and [RS00] present variants of one algorithm that also converts ladder diagrams into SMV (see [McM97] or [McM93]). The last version of the algorithm also processes timers and SFC diagrams.

The algorithm converts each rung of the ladder diagram of IEC 61131-3 standard into separated SMV module and SMV joins modules into its inner model. This method has several limitations (quoted from [RS00, page 2]):

> All contacts described in the IEC 61131-3 are taken into account, plus TON function blocks [*timer on delay*], set/reset output coils and jump instructions. [3] More precisely, we restrict ourselves to programs where
>
> (R1) All variables are boolean (e.g., no integer counter function blocks),
>
> (R2) Each LD [*ladder*] rung is composed of a testing part (test contacts and combination wires) followed by an assignment part (output coils), and
>
> (R3) There is only one LD program in the target PLC.

The algorithm APLCTRANS described in this thesis has lesser limitations that all algorithms above. Although it also supposes a PLC program with only boolean variables and without functional blocks, it is not limited to one language standard and one verification tool. Moreover, APLCTRANS can process PLC programs containing more ladders, conditional subroutine calls, conditional jumps and various timer instructions. APLCTRANS was not extended to SFC yet, but this such modification is possible.

Chambers, Holcombe and Barnard presented X-machine [CHB01], which performs functional compositions $f_5(f_4(f_3(f_2(f_1(x)))))$. But the authors did not create these compositions in such way to satisfy associative law. Thus, their algorithm requires generating trees of possible executional paths, which can result in exponential complexity.

In the contrast, APLCTRANS always runs along only one compositional path, even if the program generates 2^n paths of possible executions i.e., the composition of a PLC program with n instructions will requires at most only βn composition where $\beta \leq 3$ (see page 83).

The other works dealing with PLCs suppose that a PLC program is already converted into usable form. Their works are only distantly related to this thesis and we select only some references.

The timing analyzes of PLCs were done by Mader in [Mad00b] and by Mader and Wupper in [MW99], who also published together the overview of PLC verification problems [MW00]. Dierks published several works about his PLC-automaton including timed behavior of PLC ([Die00], [OD98], [DFMV98] and [Die97]).

[3] The previous version of the method also allowed subroutines ([RK98, page 3]).

The verifying of sequential PLC tasks can be found in [BM00] or [DCR$^+$00]. PLC graphical languages are studied by Tourlas [Tou97] and [AT98b] and by Minas [Min99], who searches semantic representation of the ladder diagrams.

Very interesting are also attempts of the opposite approach — the synthesis of error free PLC programs as [AT98a] or Moby/PLC [TD98], [DFT01].

This thesis also exploits the results from semantics of applications, logics, the theory of automata, and other works, which could be also considered as related publications, but we will cite them into the relevant sections.

Chapter 3

Abstract PLC

3.1 Model of Binary PLC

In this part we present abstract models for PLC and the automaton generated by PLC program, which create the base for further reasoning about PLCs.

The model of PLC program will suppose *only binary variables and will exclude the use of timers.* This case represents the simplest eventuality in terms of a formal analysis, nevertheless the model itself gives good theoretical background for more advanced PLC programs, even *PLCs with timers*, as we will show in Section 4.4.

Before introducing the model, we present definition of the alphabet generated by a set of binary variables. In this and following definitions, binary variables will be emphasized by membership in $\mathcal{B}$, the set of all variables having binary type. The value of a binary variable $b \in \mathcal{B}$ in a state of execution of PLC program will be denoted by $[\![b]\!]$. It always holds $[\![b]\!] = 0 \vee [\![b]\!] = 1$ for all $b \in \mathcal{B}$ where 0 represents $false$ value and 1 represent $true$ value. Using of 0 and 1 instead of boolean constants $true$ and $false$ agrees with common practice in PLC programs.

Definition 3.1 *Let A be a nonempty ordered set of binary variables $A = \{a \mid a \in \mathcal{B}\}$, then the set $\alpha(A)$ defined by Cartesian product $\{0,1\}^{|A|}$ is called* alphabet generated by *A.*

Alphabet $\alpha(A)$ contains n-tuples of all possible combinations of the values that can be assigned to variables in A. The ordering of A relates each variable to its bit in the alphabet $\alpha(A)$, whose cardinality of equals to 2^n where $n = |A|$.

Using $\alpha()$, we can define the model of a simple PLC program depicted in Figure 3.1, in which three disjoint finite sets of variables are distinguished: inputs Σ, outputs Ω, and internal variables V. The conjunction of the variable sets $\Omega \cup V$ remembers a current state of PLC execution. Notice,

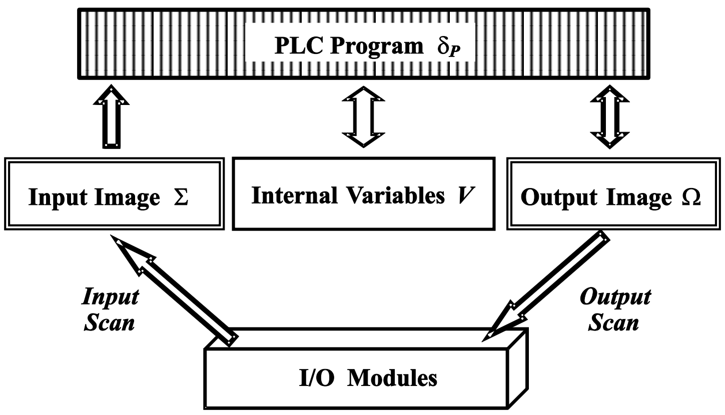


Figure 3.1: Structure of Binary PLC

that we suppose writing to the output image in the figure, because it is ordinary memory in PLCs.

However, over against entrenched practice for PLC models found in the literature, we will base the definition only on PLC variables $S = \{\Sigma, V, \Omega\}$. This approach emphasizes the facts that program's states are unknown at the beginning of an analysis and the program usually does not recognize all possible combinations of inputs.

Definition 3.2 (Binary PLC) *Binary PLC is the tuple*

$$BPLC \overset{df}{=} \langle \Sigma, \Omega, V, A_\Sigma, \delta_P, q_0 \rangle \tag{3.1}$$

where is

Σ	-	*a nonempty finite ordered set of binary inputs i.e., the input image*
Ω	-	*a nonempty finite ordered set of binary outputs i.e., the output image*
V	-	*a nonempty finite ordered set of binary internal variables of PLC*
A_Σ	-	*a nonempty subset $A_\Sigma \subset \alpha(\Sigma)$ of recognized inputs;*
δ_P	-	*PLC program described as partial mapping:* $\delta_P(q, x) : \alpha(V) \times \alpha(\Omega) \times A_\Sigma \rightarrow \alpha(V) \times \alpha(\Omega)$ *where $q \in \alpha(V) \times \alpha(\Omega)$, and $x \in A_\Sigma$*
q_0	-	*an initial state of PLC program $q_0 \in \alpha(V) \times \alpha(\Omega)$*

Nevertheless, no method capable of verifying propositions for our binary PLC exists. Known model checkers are mainly based on searching through the state space of an automaton. To analyze binary PLC, we must first unfold it to an automaton.

Definition 3.3 (Automaton generated by binary PLC)
Let $BPLC = \langle \Sigma, \Omega, V, A_\Sigma, \delta_P, q_0 \rangle$ be a binary PLC then the automaton generated by BPLC is the tuple

$$\mathbf{M}(BPLC) \overset{df}{=} \langle X, Y, Q, \delta, q_0, \omega \rangle \tag{3.2}$$

where is

X	-	*input alphabet,* $X = \alpha(\Sigma)$
Y	-	*output alphabet,* $Y = \alpha(\Omega)$
Q	-	*set of the states,* $Q = \alpha(V) \times Y$
δ	-	*transition function defined as follows:*

$$\delta(q,x) = \begin{cases} \delta_P(q,x) & : \textit{ for all } \langle q,x \rangle \in Q \times X \\ & \quad \textit{such that } \delta_P \textit{ is defined} \\ q & : \textit{ otherwise} \end{cases}$$

$q_0 \in Q$	-	*an initial state of the automaton;*
ω	-	*output function* $\omega(\langle v, y \rangle) = y$, *where* $\langle v, y \rangle \in Q$ *and* $y \in Y$ *and* $v \in \alpha(V)$

The automaton belongs to Moore's family because Y depends only on momentary state. The number of possible internal states of the automaton in Definition 3.3 is the Cartesian product $\alpha(V) \times \alpha(\Omega)$, which gives $2^{|\Omega|+|V|}$ states theoretically — we will derive in the further sections better assumption for the count of the states.

In all cases, the states rapidly grow with increasing the number of variables — well-known state explosion problem. The reductions techniques are necessary. Some of them, outlined in [BBF$^+$01] or [CL99], allow reducing the state space. For example, the set of all reachable states from given initial state q_0 use to be smaller than theoretical value $2^{|\Omega|+|V|}$ and removing such unreachable states simplifies the automaton.

This simplification method and others have advantages and drawbacks, of which the most serious is the fact that they do not often assure any feasible reduction because many of them are NP-complete problems [1] and their computability depends on an automaton in question.

In logical terms, simplifying the automaton generated by a PLC program represents impractical process — a complex automaton is first created and then laboriously reduced. Analyzing the PLC program offers an alternate solution, which could have lesser computational complexity. This approach aims at finding out the conditions, under which simplified automata can be directly generated by PLC programs.

Another problem is Definition 3.2 that supposes a PLC program describable by a deterministic function (mapping) δ_P. This assumption does not

[1]NP-complete stands for "nondeterministic polynomial time" where nondeterministic is possibility about guessing a solution. A problem is in NP if you can quickly (in polynomial time) test whether a solution is correct without worrying about how hard it might be to find the solution. If we could guess the right solution, we could then quickly test it.

hold for all programs in general. For example, if we utilize a random generator or another element with non-deterministic behavior in a PLC program, then the program will be described by a stochastic automaton with transition probability [CL99]. The termination conditions of loops or recursive functions are other serious questions.

Therefore, we must first specify a family of programming languages, for which such assumption holds, before reasoning about simplifying the automaton.

3.2 Abstract PLC Machine

In this section, we present *abstract PLC machine* (APLC machine) that creates a universal base for modelling PLCs and converting their programs. It will also support the main parts of this chapter, Sections from 3.3 to 3.4 concerning the transfer sets theory.

Readers, who are only interested in the conversion of PLC programs, can overlook this section and concentrate on the next part (Section 3.3 on page 52).

APLC machine structure follows from analysis of many PLC models of different producers as Allen-Bradley (Rockwell Automation), Siemens, and Omron. Several PLC families were studied for few years before outlining the concept of the machine.

Therefore, we will not explain the reasons which have led to creating APLC machine structure as it is. If we should give all details, we would have to overview instructions listed (or also unlisted sometimes) in thick manuals of many PLC processors. Such study might have been interesting for a PLC specialist, but unreadable for anyone else probably.

Thus, we will concentrate only on theoretical questions. We hope that the suitability of APLC machine will follow from many given propositions, mainly from Proposition 4.2 on page 102 *that a binary PLC can be modelled by an automaton of Mealy's family if and only if PLC program can be expressed as an APLC program.*

APLC machine universality will be supported by several examples, as Examples 3.11 (page 79) and 4.2 (page 112), and also Example 4.4 (page 130), in which we outline processing timers.

We present APLC machine definition in the operational semantics, which will be partially based on Nielson's machine [NN99], but we will extend it with the dump introduced in SECD machine of Ladin, described in [Gun92] or [Ram99]. [2]

[2]In the operational semantics, we are concerned with how to execute programs, how the states are modified during the execution of the statements. The abstract machine provides the implementation of a formal specification of the semantics of a programming language and gives possibility to argue about the behavior of a PLC program. The term

The operational semantics is precisely selected because PLC programs often appear as implementations of abstract machines. A sophisticated semantics with a higher abstraction, for instance denotational semantics or axiomatic semantics, could simplify reasoning about the behavior of programs but simple direct conversion rules for the instruction sets of PLCs, as those presented in Section 3.4, could not be given and converting real PLC program is the main goal of our approach presented thereinafter.

First we list the various syntax categories and give a meta-variables that will be used to range over constructs of each category. For our APLC machine the meta-variables and categories, which represent terminal symbols, are as follows:

b	will range over binary variables, $b \in \mathcal{B}$;
$bexp$	will range over boolean expressions, $bexp \in Bexp^{+}$ where $Bexp^{+}$ is language generated by $Gbexp$ grammar;
c	will range over all allowed sequences of instructions.

The meta-variables can be primed or subscripted. So, for example, b, b', b_1, b_2 all stands for binary variables. We will also suppose similarity between boolean and binary type and mutual automatic conversions between them defined by equations $0 \equiv false$ and $1 \equiv true$ (see also page 31). The structure of the construct is:

$$\begin{array}{lcl} Gbexp & ::= & 1 \mid 0 \mid b \mid \neg b \mid \neg(Gbexp) \\ & & \mid (Gbexp_1 \wedge Gbexp_2) \mid (Gbexp_1 \vee Gbexp_2) \\ & & \mid (Gbexp_1 \equiv Gbexp_2) \mid (Gbexp_1 \not\equiv Gbexp_2) \end{array}$$

where ( ) denotes ordinary parentheses for assessing the priority of boolean operations.

To specify an APLC machine, we start from its configuration in the form $\langle C, f_{reg}, E, S, D \rangle$ where

- C is code, instructions to be executed,
- f_{reg} is flag register, $f_{reg} \in \mathcal{B}$,
- E is the evaluation stack for boolean values in f_{reg},
- S is storage, and
- D is dump that is either empty or it is a four tuple $\langle C', f'_{reg}, E', D' \rangle$, where C' is a code, f'_{reg} is a flag, E' is an evaluation stack, and D' is another dump.

'machine' ordinarily refers to a physical device that performs a mechanical function. The term 'abstract' distinguishes a physically existent PLC and its programming language from one that is build for theoretical analysis — C. Gunter [Gun92].

Flag register f_{reg} is needed for evaluating boolean operations in PLCs. Dump D represents a form of 'stack', in which are remembered whole configurations. The dump allows calling subroutines.

The code is a *list* C of instructions. Lists used in this thesis can be formally defined as an ordered row $C = c_1 :: c_2 :: \ldots :: c_{n-1} :: c_n$ of n elements where :: is linking operator. In terms of data structures of computer programming, each list is a linked list with n nodes, each node contains one element and one link to next node, besides the last element that has no link.

Notice that, in terms of implementation, there is no major difference between specifying the next point of an execution by a label or by a sequence of instructions.

List are usually characterized by a pointer to their beginning i.e, by one variable habitually with size comparable to an integer variable. Therefore, specifying a point in memory by list or by label can be considered as fully equipollent.

Hence, *we may chose the first eventuality because the list describe better APLC machine operations. We select the labels for APLC language in the next section.*

Any list can have arbitrary number of elements. Let us write ν for an empty list. [3] We will also consider an element as the list with one element, but we will distinguish between lists and elements by denoting the lists with (possibly) arbitrary number of elements by upper-case letters and elements (i.e., lists containing exactly one element) by lower-case letters.

Definition 3.4 *Let C be a list. A sublist C_s is any middle part of the list $C = C_h :: C_s :: C_t$. C_h is called* prefix of C *and C_t is called* tail of C. *If the prefix contains only one element i.e., $C = c_h :: C_s :: C_t$, then it is called* head of C .

Any component can be equal to ν, so c_h, C_h, C_s, C_t are all sublists and C_h is besides prefix and C_t tail of list C. For simplification of notation, we can define $\in$ relation operator for list.

Definition 3.5 *Let C be a list and c an arbitrary element. If C can be decomposed into the form $C = C_h :: c :: C_t$, then let us write that by $c \in C$, otherwise $c \notin C$.*

The evaluation stack E is used for evaluating boolean expressions and stores values of f_{reg}. Formally, it is a list of values $E \in \{0,1\}^*$, where $\{0,1\}^*$ denotes *Kleene-closure* (see page 139).

Current values of all variables and expressions depend on momentary state of storage S. Therefore, we will use notation $[\![b]\!]\,S$ or $[\![bexp]\!]\,S$ to emphasize that values of b variable or $bexp \in Bexp^+$ expression were evaluated with respect to a momentary content of S.

[3]We hope that using ν for empty list instead of usual 'nil' does not confuse a reader.

Assignment operations change values in S. We will use the common notation $S[b \mapsto \llbracket bexp \rrbracket S]$ for assigning b the result of *bexp* expression. Expression *bexp* is first evaluated with respect to storage state S and then the new storage containing changed variable b is returned. The simplified notation $S[b \mapsto n]$ describes assigning b a constant value n.

Binding the result of evaluation to the storage prevents misinterpretations arising from a different evaluation order of variables. For example, let $\llbracket b \rrbracket S$ be 0. *We will always suppose* that after executing $S[b \mapsto \llbracket \neg b \rrbracket S]$ the formula $\llbracket b \rrbracket S$ *again equals to* 0, because the both evaluation of b are based on the same storage content S. To change the second result, we must explicitly write $S := S[b \mapsto \llbracket \neg b \rrbracket S]$. Now the content of S has changed and $\llbracket b \rrbracket S$ equals to 1.

Example 3.1

If $S = \{a = 1, b = 0\}$ then $\llbracket a \rrbracket S$ equals to 1. $S_1 := S[a \mapsto \llbracket b \rrbracket S]$ yields new storage $S_1 = \{a = 0, b = 0\}$. Now $\llbracket a \rrbracket S$ equals again to 1, but $\llbracket a \rrbracket S_1$ equals to 0. This operation recommends some assignment $a := b$. But assignments are formulas over some abstract variables, whereas $S[a \mapsto \llbracket b \rrbracket S]$ is the directive how this assignment should be evaluated and where are taken data from.

The semantics of the abstract machine is given by the operational semantics and specified by a transition system. The configuration have the form $\langle C, f_{reg}, E, S, D\rangle$ as described above and transition relations $\triangleright$ show how to execute the instructions:

$$\langle C, f_{reg}, E, S, D\rangle \triangleright \left\langle C', f'_{reg}, E', S', D'\right\rangle \tag{3.3}$$

The idea is that one step of execution will transform the configuration $\langle C, f_{reg}, E, S, D\rangle$ into $\left\langle C', f'_{reg}, E', S', D'\right\rangle$. The relations are defined by the axioms of Table 3.1, in which instruction codes are emphasized by Caps.

Example 3.2

We demonstrate APLC machine on simple operation that could be written as Pascal-like command 'x:=(x AND y);'. Beginning configuration H will be transformed (we use [ ] to emphasize the elements of C list) as:

$$\begin{aligned}
H \;=\;& \left\langle \begin{array}{l} C = [\textsc{Load}\ x] :: [\textsc{Or}\ y] :: [\textsc{Store}\ x],\ f_{reg} = 1, \\ E = \nu,\ S = \{x = 0, y = 1\},\ D = \nu \end{array} \right\rangle \\
\triangleright\;& \langle [\textsc{Or}\ y] :: [\textsc{Store}\ x], f_{reg} = \mathbf{0}, E = \nu, S = \{x = 0, y = 1\}, D = \nu\rangle \\
\triangleright\;& \langle [\textsc{Store}\ x], f_{reg} = \mathbf{1}, E = \nu, S = \{x = 0, y = 1\}, D = \nu\rangle \\
\triangleright\;& \langle \nu, f_{reg} = 1, E = \nu, S = \{x = \mathbf{1}, y = 0\}, D = \nu\rangle + \textit{Terminate}
\end{aligned}$$

Initialization of evaluation		
$\langle \textsc{Init} :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, 1, \nu, S, D\rangle$
Operation with flag register		
$\langle \textsc{Load}\ bexp :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, [\![bexp]\!]\,S, E, S, D\rangle$
$\langle \textsc{And}\ bexp :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, f_{reg} \wedge [\![bexp]\!]\,S, E, S, D\rangle$
$\langle \textsc{Or}\ bexp :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, f_{reg} \vee [\![bexp]\!]\,S, E, S, D\rangle$
$\langle \textsc{Not} :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, \neg f_{reg}, E, S, D\rangle$
$\langle \textsc{TAnd} :: C, f_{reg}, top :: E, S, D\rangle$	$\rhd$	$\langle C, f_{reg} \wedge top, E, S, D\rangle$
$\langle \textsc{TAnd} :: C, f_{reg}, \nu, S, D\rangle$	$\rhd$	$\langle C, 0, \nu, S, D\rangle$
$\langle \textsc{TOr} :: C, f_{reg}, top :: E, S, D\rangle$	$\rhd$	$\langle C, f_{reg} \vee top, E, S, D\rangle$
$\langle \textsc{TOr} :: C, f_{reg}, \nu, S, D\rangle$	$\rhd$	$\langle C, f_{reg}, \nu, S, D\rangle$
Assignments		
$\langle \textsc{Store}\ b :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, f_{reg}, E, S[b \mapsto f_{reg}], D\rangle$
$\langle \textsc{Set}\ b :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, f_{reg}, E, S[b \mapsto (f_{reg} \vee [\![b]\!]\,S)], D\rangle$
$\langle \textsc{Res}\ b :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, f_{reg}, E, S[b \mapsto (\neg f_{reg} \wedge [\![b]\!]\,S)], D\rangle$
Rising and falling edge detection		
$\langle \textsc{REdge}\ b :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, f_{reg} \wedge [\![\neg b]\!]\,S, E, S[b \mapsto f_{reg}], D\rangle$
$\langle \textsc{FEdge}\ b :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C, \neg f_{reg} \wedge [\![b]\!]\,S, E, S[b \mapsto f_{reg}], D\rangle$
Program control (*note: brackets* [ ] *only emphesize the heads of codes*)		
$\langle [\textsc{Jp}\ C_{new}] :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C_{new}, f_{reg}, E, S, D\rangle$
$\langle [\textsc{Jpc}\ C_{new}] :: C, 1, E, S, D\rangle$	$\rhd$	$\langle C_{new}, 1, E, S, D\rangle$
$\langle [\textsc{Jpc}\ C_{new}] :: C, 0, E, S, D\rangle$	$\rhd$	$\langle C, 0, E, S, D\rangle$
$\langle [\textsc{Js}\ C_{new}] :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle C_{new}, 0, \nu, S, \langle C, f_{reg}, E, D\rangle\rangle$
$\langle [\textsc{Jsc}\ C_{new}] :: C, 1, E, S, D\rangle$	$\rhd$	$\langle C_{new}, 0, \nu, S, \langle C, 1, E, D\rangle\rangle$
$\langle [\textsc{Jsc}\ C_{new}] :: C, 0, E, S, D\rangle$	$\rhd$	$\langle C, 0, E, S, D\rangle$
$\langle \textsc{End} :: C, f_{reg}, E, S, D\rangle$	$\rhd$	$\langle \nu, 1, \nu, S, D\rangle$
Loading dump		
$\left\langle \nu, f_{reg}, E, S, \langle C', f'_{reg}, E', D'\rangle\right\rangle$	$\rhd$	$\left\langle C', f'_{reg}, E', S, D'\right\rangle$
Termination condition		
$\langle \nu, f_{reg}, E, S, \nu\rangle$	$\rhd$	$\langle \nu, 1, \nu, S, \nu\rangle$ + *terminate*
Manipulations with evaluation stack		

Table continues in Table 3.2

Table 3.1: Operational Semantics for APLC machine - Part I.

Table continues from Table 3.1

Manipulations with evaluation stack

$\langle \text{PUSH} :: C, f_{reg}, E, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, f_{reg} :: E, S, D\rangle$
$\langle \text{EPUSH } \mathit{bexp} :: C, f_{reg}, E, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, \mathit{bexp} :: E, S, D\rangle$
$\langle \text{POP} :: C, f_{reg}, \mathit{top} :: E, S, D\rangle$	$\triangleright$	$\langle C, \mathit{top}, E, S, D\rangle$
$\langle \text{POP} :: C, f_{reg}, \nu, S, D\rangle$	$\triangleright$	$\langle C, 0, \nu, S, D\rangle$
$\langle \text{DUP} :: C, f_{reg}, \mathit{top} :: E, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, \mathit{top} :: \mathit{top} :: E, S, D\rangle$
$\langle \text{DUP} :: C, f_{reg}, \nu, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, \nu, S, D\rangle$
$\langle \text{DROP} :: C, f_{reg}, \mathit{top} :: E, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, E, S, D\rangle$
$\langle \text{DROP} :: C, f_{reg}, \nu, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, \nu, S, D\rangle$
$\langle \text{FSWP} :: C, f_{reg}, \mathit{top} :: E, S, D\rangle$	$\triangleright$	$\langle C, \mathit{top}, f_{reg} :: E, S, D\rangle$
$\langle \text{FSWP} :: C, f_{reg}, \nu, S, D\rangle$	$\triangleright$	$\langle C, 0, f_{reg} :: \nu, S, D\rangle$
$\langle \text{ESWP} :: C, f_{reg}, \mathit{top} :: \mathit{tnx} :: E, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, \mathit{tnx} :: \mathit{top} :: E, S, D\rangle$
$\langle \text{ESWP} :: C, f_{reg}, \nu, S, D\rangle$	$\triangleright$	$\langle C, f_{reg}, \nu, S, D\rangle$

Table 3.2: Operational Semantics for APLC Machine - Part II.

where logical constants **1** or **0** written in bold font indicate changed values by last step.

Table 3.1 does not present a minimal set of instructions. Apart from using *bexp* $\in$ *Bexp*$^+$ instead of a step-by-step evaluation of boolean expressions, there are defined the operations that could be composed from primitives. For instance, LOAD *bexp* is equivalent to " OR 1 ; AND *bexp*;"

This extended instruction set gives possibility for the optimizations of APLCTRANS algorithm for the conversion of PLCs, that will be presented in Section 3.4 (see page 73).

The real implementation of APLC machine can use a subset of the listed operations. If we prove all theorems in this chapter for the extended set of instruction they will certainly hold for every its subset.

Definition 3.6 *Let $H = \langle C, f_{reg}, E, S, D\rangle$ be an APLC configuration. If $C \neq \nu$ then* top instruction of H *is the instruction c_a given by $C = c_a :: C_2$, otherwise* top instruction of H *equals ν. Similarly, if $C_2 \neq \nu$, then* next instruction of H *is the instruction c_b defined as $C = c_a :: c_b :: C_3$ or $C_2 = c_b :: C_3$, otherwise* next instruction of H *equals to ν.*

Top instruction is also the head of list C.

Definition 3.7 (APLC step) *Let $H = \langle C, f_{reg}, E, S, D\rangle$ be an APLC configuration. Transforming H into $H' = \left\langle C', f'_{reg}, E', S', D'\right\rangle$ (denoted as $H \rhd H'$), by the application of one axiom from Table 3.1, is called* APLC machine step *(abbreviated* APLC step*). H' is* successor of H*, denoted as $succ(H)$. Let c_a be* top code of H *satisfying $c_a \neq \nu$ then the* APLC step *is called* APLC instruction step.

Definition 3.8 (APLC scan) *Let H_0 be a configuration $\langle C_0, 1, \nu, S_0, \nu\rangle$, where C_0 is an initial code and S_0 represents an initial content of storage. Let* $\mathrm{SC}_{H_0} = H_0 :: H_1 :: \ldots :: H_n$ *be the list of sequential configurations, abbreviated as* sequence*, in which $H_{i+1} = succ(H_i)$ for all $0 \leq i < n$, then* SC_{H_0} *is called* APLC scan *of configuration H_0. Moreover, if* SC_{H_0} *is finite and $H_n = \langle \nu, 1, \nu, S_n, \nu\rangle$ then* SC_{H_0} *is called* terminal APLC scan.

Example 3.2 has performed 3 instruction steps before the termination. Thus H has terminal APLC scan.

We will reduce all further considerations to deterministic APLC machines which allow to predict results of APLC steps with certainty i.e., any their configuration contains all necessary information for unambiguous calculation of its successor.

Definition 3.9 *APLC machine is call* deterministic APLC machine *if set* $\mathrm{SC}^+_H = \{\mathrm{SC}_H \mid \mathrm{SC}_H$ *is a terminal APLC scan of H*$\}$ *contains one element.*

Although the requirement above could look as a triteness at first glance we will show in the following paragraphs that it is too strong to be practically satisfied, because deterministic APLC machine must not contain any element with stochastic behavior.

An APLC machine will be non-deterministic, for instance, if an overflow of its evaluation stack happens during APLC scan of H and this event either terminates the scan with an exception, in that case SC^+_H will be empty, or adds a random value to the stack so there will be more possible APLC scans of H.

Lemma 3.1 *APLC machine is deterministic if it satisfies the following:*

- *its evaluation stack has infinite length,*
- *the number of possible dumps is unlimited, and*
- *evaluations of expressions and manipulations with storage are always unambiguous and they never abort APLC scan.*

Proof: The proof has two parts. First we analyze possible sources of unexpected termination, then we prove uniqueness of APLC scan by contradiction. The first part of the proof is based on Definition 3.9 and Table 3.3, and also on the understandable assumption that code C contains only defined codes, otherwise discussions about any behavior of the program have no sense.

APLC instructions do not allow any direct manipulations with the dumps or program memory with exception of variables, so they can never accidently jam instruction codes. The definition rule out the interruptions of program by the evaluation of some inconsistent expression ($e \notin \mathit{Bexp}^+$) and the overflow evaluation stack S or dumps i.e., we suppose that there will be always enough place in the dump and S to add new values.

The analysis of operations loading top of the evaluation stack (Pop, FSwp, TAnd, and TOr) shows that APLC machine recognizes stack underflows i.e., the attempts of reading values from empty stack, and it has always deterministic behavior in these situations. Dump underflows are impossible because the dump D is loaded when code C is empty, and the concurrent emptiness of the dump and code terminate scans regularly.

Now we must prove that SC^+_H contains exactly one sequence of configurations under the assumptions described above. SC^+_H is certainly non-empty — it contains at least one sequence beginning with H. Let, on a contrary, be H an APLC configuration whose SC^+_H contains more sequences. If $\mathrm{SC}, \mathrm{SC}' \in \mathrm{SC}^+_H$ and $\mathrm{SC} \neq \mathrm{SC}'$ then SC and SC$'$ only differ in their tails because the both sequences begin with H. We can write them as lists $\mathrm{SC} = \mathrm{SC}_h :: \mathrm{SC}_t$, $\mathrm{SC}' = \mathrm{SC}_h :: \mathrm{SC}'_t$ where $\mathrm{SC}_t \neq \mathrm{SC}'_t$.

Evidently $\mathrm{SC}_h \neq \nu$ because it begins with H, so it can be written as $\mathrm{SC}_h = \mathrm{SC}_{h_0} :: H_h$, $H_h \neq \nu$, where SC_{h_0} is some sequence and H_h end configuration of sequence SC_h. The configuration H_h should branch into two different configurations H_t and H'_t. They are prefixes of $\mathrm{SC}_t = H_t :: \mathrm{SC}_{t2}$ and $\mathrm{SC}'_t = H'_t :: \mathrm{SC}'_{t2}$, where $H_t = \mathit{succ}(H_h)$, $H'_t = \mathit{succ}(H_h)$, and $H_t \neq H'_t$.

We prove that H_t and H'_t exist and are non-empty. First we analyze the configurations with empty code. H_h differs from configurations in the form $\langle \nu, f_{reg}, E, S, \nu \rangle$, because those terminates APLC scan and cannot create two different successors.

H_h also differs from $\left\langle \nu, f_{reg}, E, S, \left\langle C', f'_{reg}, E', D' \right\rangle \right\rangle$ because those load the dump thus they have only one successor defined by a content of the dump. Inspecting other axioms in Table 3.3 we find out that their application also create successors of H_h, therefore H_t and H'_t are non-empty.

In the previous paragraph, we have ruled out all configurations with empty codes C, hence H_h will have non-empty top code c_h. Code c_h is clearly not End code. Now in Table 3.3, there remain only instruction steps of APLC machine. They transform H_h into new configuration, which is fully determined by c_h and the contents of storage and evaluation stack of H_h.

Supposing identical results for identical evaluations of assignment operation $S[b \mapsto [\![\mathit{bexp}]\!]\, S]$, $\mathit{bexp} \in \mathit{Bexp}^+$, was the primary assumption for APLC machine given in Page 37. Therefore H_t must always be equal to H'_t and we have a contradiction. $\square$

'AProgram'	::=	'Blocks'
'Blocks'	::=	'ABlock' | 'ABlock' : 'Blocks'
'ABlock'	::=	'End-instruction'
		| *label* : 'End-instruction'
		| 'AStatement'; 'ABlock'
		| *label* : 'AStatement'; 'ABlock'
'AStatement'	::=	'Op-instruction' | 'Control-instruction'
		| 'End-instruction'
'Control-instruction'	::=	'Control-code' *label*
		where 'Control-code' $\prec$ *label*
		(*see Definition 3.10 on page 42*)
'Op-instruction'	::=	Init | Load *bexp*
		| And *bexp*| Or *bexp*| Not
		| TAnd | TOr
		| Push | EPush *bexp*| Dup
		| FSwp | ESwp | Pop | Drop
		| Store *b* | Set *b* | Res *b*
		| REdge *b* | FEdge *b*
'End-instruction'	::=	End
'Control-code'	::=	Jp | Jpc | Js | Jsc
		where Instruction codes are defined
		in Table 3.1 and *bexp* $\in Bexp^+$.

Table 3.3: Grammar of APLC Language

APLC machine expresses an implementation and allows wide rage of possible programs. To make exacter propositions, the syntax of APLC language that machine executes must also be given. Its grammar is listed in BNF-style (Backus-Naur Form) in Table 3.3. Non-terminals are single quoted.

In the following definitions, we will also use non-terminals names to denote languages, which were generated from homonymous non-terminal symbol of the grammar i.e., *AProgram* will denote language generated from non-terminal symbol 'AProgram', similarly we will use *AStatement* for language generated from non-terminal symbol 'AStatement', similarly for *ABlock*.

To specify destination addresses for jumps and calls in the language, new semantic category was added to b and *bexp* (see page 35):

label is any identifier unique within one program in *AProgram*.

and it is utilized together with $\prec$ relation.

Definition 3.10 *Let L be a language and $s \in L$ any string. If s_1, s_2 are two* *substrings of s then $s_1 \prec s_2$ iff $s = a.s_1.b.s_2.c$, where a, b, c are substrings of s, otherwise $s_1 \not\prec s_2$.*

Relation $\prec$ of substrings was used in the table to determine the constraint for the labels that stand for the operands of jumps and calls. The relation violates BNF grammar rules, on the other side it simplifies APLC language definition.

In case of APLC language, $\prec$ relation reflects the order, in which statements are physically concatenated in the source code of an APLC program, and it serves for excluding jumps and calls to backward addresses in terms of some ordering of APLC statements.

Example 3.3

	LOAD	b_1;	
	JPC	*label1*;	Here, it holds that *this* JPC $\prec$ *label1*
	AND	b_2;	
label1:	STORE	b_1;	
	JPC	*label1*;	Here, *this* JPC $\not\prec$ *label1*
	END		

The relation $\prec$ will also allow to specify a simple termination condition for APLC programs and moreover, their effective composition in Section 3.4. To prove APLC termination, we must first present the following definitions.

Definition 3.11 (Compilation.) *Let $AL \in AProgram$ be some APLC program. Code C of configuration $H = \langle C, 1, \nu, S, \nu \rangle$ of APLC machine is called* compilation of AL with respect to storage S, *denoted as $AL \overset{S}{\rightharpoonup} C$, if C satisfies following rules:*

- *there exists a mapping, $m : Label \longrightarrow Storage$, assigning every variable unique location in storage S,*
- *each string generated by 'AStatement' rule is converted according to Tables 3.1 and 3.3,*
- *for any $s_1, s_2 \in AL$ two strings generated by 'AStatement' rule, it holds that then $s_1 \prec s_2$ implies $C = C_p :: c_1 :: C_s :: c_2 :: C_t$ where C_p, C_s, C_t are (possibly empty) sublists of C and c_1, c_2 are the codes which were converted s_1, s_2 to, and*
- *each $label_i$ is replaced by list C_i, $C = C_{pfx_i} :: C_i$, where C_{pfx_i} is a prefix of C and C_i begins with the code, to which was converted the statement labeled by $label_i$.*

In other word, all statements are correctly converted to APLC machine codes, their ordering in AL is preserved in C, and labels are decoded to lists.

The previous perspicuity of the compilation is rough. To create an exact definition, we should formally specify its algorithm. However, we believe that the correspondence between APLC machine and its language syntax is clear from the both tables and the omitting simple but long specifications will not ever invalidate theorems given hereafter.

Proposition 3.1 *Let $AP \in AProgram$ be a finite APLC program with $AP \stackrel{S}{\rightharpoondown} C$ compilation. If a deterministic APLC machine is used for interpreting C, then APLC scan of AP is always terminal for any content of the storage S.*

Proof: The proof proceeds by induction on the length of APLC program. The base case AP^0 is the program containing only one instruction END and with compilation $AP^0 \stackrel{S}{\rightharpoondown} C^0$. The axiom for END assigns ν to the code of APLC machine what results in certain termination for C^0 according to the last rule in Table 3.1. Therefore AP^0 always terminates after a finite number of APLC steps and the same condition evidently holds also for the set of all possible non-empty tails of C^0, which can be defined as $\text{Tails}(C^0) \stackrel{df}{=} \{C_a \mid C^0 = C_a :: C_b, C_b \neq \nu\}$. C^0 containing only End instruction has $\text{Tails}(C^0)$ with one element $\{\text{END}\}$.

The inductive step is carried out by proving the consequent from its antecedent. Let AP^i be an APLC program with compilation $AP^i \stackrel{S}{\rightharpoondown} C^i$ whose all $\text{Tails}(C^i)$ have terminal APLC scans. We create AP^{i+1} by an inserting APLC instruction before the program. The conditions in Definition 3.11 assure keeping the order of instructions so new instruction will be translated to code c placed before C^i. We obtain $C^{i+1} = c :: C^i$ and prove that C^{i+1} has again terminal APLC scan.

To prove that, we divide APLC instructions into four disjoint groups. The first trivial possibility represents END instruction — $\text{Tails}(C^{i+1})$ will have certainly terminal APLC scans.

The same will hold for the group incorporating all instructions marked in Table 3.3 as 'Op-instruction'. They perform an evaluation and create a new configuration, which has C^i as the code and changed content of flag register and evaluation stack.

Deterministic APLC machine has foreseeable scan with one possible sequence for one content of storage. Initial values of evaluation stack or flag register can change values in the storage obtained after APLC scan but they cannot alter finality of the scan, as we have discussed in Lemma 3.1. Therefore, changing initial values of evaluation stack or flag register will preserve terminal APLC scan condition for all $\text{Tails}(C^{i+1})$.

The third group consists of jump instructions JP and JPC. If flag register equals to 0 then JPC will behave like an instruction in 'Op-instruction' group, for which we have already proved terminal APLC scan. Otherwise, the jumps will create new configuration from previous one by replacing the code by their operand. According to the conditions for labels specified in Definition 3.10, the new code sequence can be only an element of Tails(C^i). All Tails(C^i) have terminal APLC scans and so inserting jump will create C^{i+1} satisfying the same condition.

The last possibility represents the instruction Js and Jsc. If flag register equals to 0 then Jsc behaves as Jpc, proved in paragraphs above, otherwise Jsc is equivalent to Js, so we can limit our reasoning only to Js.

In this proof only, let us write for the current configuration, in which is Js code evaluated, $\check{H}_0^{i+1} = \left\langle (Js\ C_{Js}^{i+1}) :: C^i, f_{reg}, E, S, \nu \right\rangle$, where C_{Js}^{i+1} denotes an operant of Js code and $C^{i+1} = (Js\ C_{Js}^{i+1}) :: C^i$. The dump is empty because now APLC scan begins with Js code inserted by this case of the induction. Js code transforms configuration $\check{H}_0^{i+1}$ into $\check{H}_1^{i+1} = \left\langle C_{Js}^{i+1}, 1, \nu, S, \left\langle C^i, f_{reg}, E, \nu \right\rangle \right\rangle$.

The both conditions for labels mentioned when proving jumps and the induction's assumption assure that $C_{Js}^{i+1} \in$ Tails(C^i). Therefore C_{Js}^{i+1} has terminal APLC scan, which consists of $\mathrm{SC}(H_1^{i+1}) = H_1^{i+1} :: H_2^{i+1} :: \ldots :: H_n^{i+1}$ where $H_1^{i+1} = \left\langle C_{Js}^{i+1}, 1, \nu, S, \nu \right\rangle$ and $H_n^{i+1} = \langle \nu, 1, \nu, S, \nu \rangle$ (Definition 3.8). This terminal APLC scan is assured for any content of storage S. We must prove that the scan remains terminal for $\check{H}_1^{i+1}$.

Comparing H_1^{i+1} with $\check{H}_1^{i+1}$ reveals that the configurations only differ in their dumps. H_1^{i+1} has its dump equal to ν. In contrast, $\check{H}_1^{i+1}$ contains the dump, which we denote by $\check{D} = \left\langle C^i, f_{reg}, E, \nu \right\rangle$.

If $\mathrm{SC}(H_1^{i+1})$ includes only codes that do not manipulate with the dump then $\mathrm{SC}(\check{H}_1^{i+1})$ will consist of the same configurations with the exception of replacing their ν dumps by $\check{D}$. H_n^{i+1} changes into $\check{H}_n^{i+1} = \left\langle \nu, 1, \nu, S, \check{D} \right\rangle$, to which now the penultimate axiom in Table 3.1 will be applied instead of the last terminating axiom, and dump $\check{D}$ will be loaded. We obtain the configuration $\left\langle C^i, f_{reg}, E, S', \nu \right\rangle$, for which we have already shown that has terminal APLC scan when proving 'Op-instruction'.

Now, situations only remained, in which the dump is changed during a subroutine evaluation. We can immediately exclude the last axiom in Table 3.1, because it can be applied only at the end of $\mathrm{SC}(H_1^{i+1})$. The penultimate axiom can be also crossed out because if a configuration satisfied condition for loading dump and did not terminate sequence $\mathrm{SC}(H_1^{i+1})$, there must exist an associated configuration, which created this loaded dump i.e., the configuration with Js ot Jsc. Similar consideration can be also given for the remaining cases — codes End, Js, and Jsc.

We have proved that AP^{i+1} has terminal APLC scan and because any program can be created by inserting instructions each *AProgram* has terminal APLC scan if deterministic APLC machine is used. □

3.2.1 Discussion

Proving terminal scans for every *AProgram* was possible due to the syntax of APLC language, which excludes loops and recursive procedures. In this part, we will consider these limits and also practical usability of the terminal scan proposition.

Loops.

The absence of loops in *AProgram* could weighty reduce any usability of most programming languages, but PLC programs contain loops rarely. There are three main reasons not to program them in PLCs:

- A troubleshooting of a running technology is more difficult if PLC program contains loops,
- loops can prolong PLC scan whose time is strictly limited by the requirements for regular updating of inputs and outputs, and
- cyclic PLC behavior encloses each $AP \in AProgram$ inside the endless loop maintained by PLC scan and AP is executed in the form similar to a Pascal program:

  ```
  repeat
  AP;
  until false;
  ```

  Thus any loop in AP can be replaced by a new algorithm, which performs one evaluation of the operations inside the loop during every PLC scan. This modification does not significantly prolong PLC scan and always exists for any **for**, **repeat** ... **until**, or **while** loops of the structural programming languages.

Therefore, PLC programmers are mostly asked not to create loops. [4] Naturally, some operations require a looping to keep code simple because their primary substance is based on cycling, like initializations or moving data blocks. For such cases, PLCs usually contain macro instructions capable of programming these manipulations by one command. [5] These instruction have clear termination conditions and can be mostly decomposed into sequences of APLC instructions. For instance, moving variables is equivalent to several assigning operations.

[4] The author of the thesis had cooperated on writing industrial PLC programs, in which were forbidden not only loops, but also jumps or subroutines with conditional execution i.e., the instruction with behavior similar to APLC codes Jsc or Jpc.

[5] For instance, PLC type 'PLC 5' offers also special 'file instructions' that also allow providing data manipulations asynchronously with PLC scan [Roc98, pg. 109].

Recursive subroutines.

APLC programs do not allow recursive calls as most of manufactured PLCs so this limit does not reduce primary purpose of APLC programs, which were presented mainly as the universal abstraction for expressing PLC programs.

Besides, we could present nearly similar considerations on calls as those discussed for jumps. Therefore, subroutines are mainly used for organizing PLC programs into smaller blocks. Conditional calls are programmed sporadically.

Complexity of Terminal Scan.

We have proved that APLC programs have terminal APLC scans. The question we will now ask is how many APLC instruction steps will be done before finishing APLC scan.

Lemma 3.2 *Let $AP \in AProgram$ be an APLC program. If the number of instructions in AP equals to n, then scan of AP will at most consist of* $2^n - 1$ *instruction steps.*

Proof: We prove lemma by the construction of worst case APLC program. Any APLC program must contain at least 1 instruction End according to syntax (see Table 3.3 on page 42), so $n \geq 1$. Instruction Js is the only one capable of increasing the number of APLC steps.

We create program as long sequence of n instructions Js, which will be terminated by End instruction as required by APLC syntax. Every Js calls its next Js instruction as shown in Table 3.4 (see page 48).

First n instructions Js adds one execution of itself and induces double execution of the following instructions. The last End induces two steps — clearing the code and loading the dump but loading dump is not counted as an instruction step (Definition 3.7). Thus total number of instruction steps is:

$$\underbrace{1 + 2(1 + 2(\ldots(1 + 2(1 + 2)\ldots))}_{n-1 \text{ addition operators}} = \sum_{i=1}^{n-1} 2^i \;= 2^n - 1 \text{ where } n \geq 1 \qquad (3.4)$$

where inner term $(1 + 2)$ includes the both last Js and End instruction □

The result of the lemma above is overwhelming and devalues Proposition 3.1. But manufactured PLCs allow usually limited number of nesting calls. In this case the number of instruction steps is smaller, but still huge.

		Js $label_1$
$label_1$	:	Js $label_2$
$label_2$	:	Js $label_3$
		...
$label_{i-1}$	:	Js $label_i$
$label_i$	:	Js $label_{i+1}$
		...
$label_{n-2}$	:	Js $label_{n-1}$
$label_{n-1}$	:	End

Table 3.4: Worst Case APLC Program

$$
\begin{array}{lcl}
 & & \left.\begin{array}{ll} \textsc{Js} & label_1 \\ \textsc{Js} & label_1 \\ \ldots & \\ \textsc{Js} & label_1 \end{array}\right\} m_1 \\
 & & \textsc{End} \\
label_1 & : & \left.\begin{array}{ll} \textsc{Js} & label_2 \\ \textsc{Js} & label_2 \\ \ldots & \\ \textsc{Js} & label_2 \end{array}\right\} m_2 \\
 & & \textsc{End} \\
 & & \ldots \\
label_{k-1} & : & \left.\begin{array}{ll} \textsc{Js} & label_k \\ \textsc{Js} & label_k \\ \ldots & \\ \textsc{Js} & label_k \end{array}\right\} m_{k-1} \\
label_k & : & \left.\begin{array}{l} \text{TAnd} \\ \text{TOr} \\ \ldots \\ \text{TAnd} \end{array}\right\} m_k \times \text{ any 'Op-instruction'} \\
 & & \textsc{End}
\end{array}
$$

Table 3.5: Worst Case APLC Program with Nesting Limit

Lemma 3.3 *Let $AP \in AProgram$ be an APLC program with n instructions different from instruction steps* End *and satisfies the limit of at most m nested calls. If we denote by $k = \min(m+1, \frac{n}{e})$, where e stands for Euler's constant, then APLC scan of AP will contain at most $\left(\frac{n}{k}\right)^k$ instruction steps different from instructions steps* End*,* Js*, and* Jsc*.*

Proof: Using similar approach as in the previous lemma we create the worst case APLC program. To satisfy limit of nested calls, we divide the program into k blocks. Each block $1 \leq i < k$ contains m_i instructions that call the next block and are terminated by one End with the exception of $k-1$ block that continues to k block. Last block k consists of m_k instructions belonging to 'Op-instruction' group.

We are only interested in rough approximation now. To obtain easy solution, we solve equations in real domain and consider only the steps done in the last block with m_k 'Op-instruction' i.e., the instruction providing manipulation with data. APLC steps of End and Js will not be counted. We will consider them in Proposition 3.2 given below.

The program has structure shown in Table 3.5. Its 1st block invokes 2nd block m_1 times, that calls 3rd block m_2 times and so on, until the last block executes m_k instructions. The total steps are determined by the equation $\prod_{i=1}^{k} m_i$ with the constraint $n = \sum_{i=1}^{k} m_i$. By substituting the constraint and using partial derivatives with respect to m_i, we find out that the equation reaches the extreme if $m_i = m_j$ for all i, j.

Therefore, the number of steps is equal to $\left(\frac{n}{k}\right)^k$. The value of the function is growing with increasing k, the first derivative with respect to k is positive for $0 < k < \frac{n}{e}$, where e denotes Euler's constant. The point $k = \frac{n}{e}$ is the extreme but k cannot exceed $m+1$ (limit of m nested calls plus the operations at the last block), so we use $k = \min(m+1, \frac{n}{e})$. □

We summarize the both previous lemmas in the proposition:

Proposition 3.2 *The number of steps of APLC program with n instructions is EXPTIME task with complexity $O(2^{n-1})$. If nested calls are limited to m levels then the task turns to P-problem with complexity $O(n^{m+1})$.* [6]

Proof: We have already proved in Lemma 3.2 that solving number of APLC program steps has exponential complexity. In Lemma 3.3, we show that limiting nested calls has polynomial complexity if we take in account only

[6]The terms are taken from computational complexity theory. EXPTIME means a problem that can be solved in exponential time. P-problems can be solved in polynomial time. O() is Landau's symbolism named after the German number theoretician Edmund Landau who invented the notation for describing the asymptotic behavior of functions. Big O tells us at which order a function grows or declines. If c is a constant then $O(1)$ means a constant, $O(n)$ linear , $O(n^c)$ polynomial, and $O(c^n)$ exponential complexity.

’Op-instruction’ instruction steps. So we must only prove that polynomial complexity holds also for all steps.

We use the same worst case program shown in Table 3.5, in which $k = m+1$ (the limit of nested calls). The program will be divided into blocks with lengths $m_1, m_2, \ldots, m_{m+1}$ that need not be uniform — this extreme exists only under the simplified conditions in Lemma 3.3.

Each m_i can be expressed as a fraction of n, $m_i = \beta_i n$, where β_i are real numbers and $\sum_{i=1}^{m+1} \beta_i = 1$ because $\sum_{i=1}^{m+1} m_i = n$.

Blocks $1 \le i \le m$ add to the total number of steps $\beta_i n$ instructions JS, one END and $\beta_i n$ executions of the following block. The last block adds only itself i.e., $\beta_{m+1} n$ steps. The number of steps is equal to:

$$\underbrace{1+\beta_1 n+\beta_1 n(1+\beta_2 n+\beta_2 n(\ldots(1+\beta_m n+\beta_m n(\beta_{m+1} n))\ldots))}_{m+1\ ()\ \text{operations}} \tag{3.5}$$

All β_i are real constants so the equation can be rewritten into a polynomial, whose term with highest power equals to $n^{m+1} \prod_{i=1}^{m+1} \beta_i$. Therefore, the complexity of the task is polynomial with $O(n^{m+1})$.

□

The nesting limit varies with PLC type. For example, Allen-Bradley PLC-5 family of PLCs allows only 8 nesting subroutines [Roc98, pg. 178] and the same limit has also Siemens S7-200 family of PLCs [Sie02, pg. 203], but it is not a general value certainly. If we substitute this limit into Lemma 3.3 we obtain formula $\left(\frac{n}{9}\right)^9$ for $n > 9e$. The value equals to 9846 for a program with 25 instructions different form END (25 is first integer value satisfying condition $n > 9e$). The real number of steps will be slightly less because the sizes of the blocks must be integer values so the expression does not reach its extreme.

The count of steps growing with 9th power will easily cause the state explosion problem. If we concentrate on programs satisfying lesser nested limit we will obtain better complexity condition.

The other important limit of real PLC program is automatic check of PLC scan by external timer, which is commonly called ”watch dog”. This feature is probably built in every manufactured PLC. [7]

The watch dog measures time passed from the last updating inputs and outputs (I/O scan). If PLC processor does not begin new I/O scan before elapsing predefined timer constant, PLC program is stopped and a watch dog error is announced. So the watch dog supervises the cyclic behavior of PLC scan.

If we suppose analyzing a program designed for controlling a technology, not for creating a contraindication, then the program could probably have

[7] The author does not know any PLC without a watch dog done by some method.

a terminal scan with a 'reasonable' value of steps. [8]

To assure that terminal scan has generally lower complexity, the set of APLC instruction must be either reduced (deleting jumps and calls also removes all problems) or extended by the constraints, which will prohibit building program constructions having higher complexity in the term of analysis. However, this approach could complicate converting PLC program to APLC language, therefore we will prefer higher universality even if it is paid by weaker computability propositions.

Moreover, *we will show in the further sections that the number of instruction steps is not the main criterion.* The conversion of an APLC program, which has with n instructions and $2^n - 1$ instruction steps, into automaton can still have $O(n)$ complexity under some hopeful conditions.

[8]This raises certainly the question how big could be this 'reasonable' value? Let us assume that common PLC scan takes around 50 ms (i.e. sample rate 20 Hz of I/O) and ordinary PLC processor could execute 2000 bit instruction during 1 millisecond on average. We obtain 100000 instruction steps. The number represent raw informative value for worst case scan. In practice, we can expect much lesser values, because complex instructions, for example subroutines, take more time (PLCs are relatively slow computers.)

3.3 Transfer Sets

3.3.1 Overview of Goal

To express APLC program scan as a mapping δ_P according Definition 3.2, we need to convert APLC instructions to proper forms, which can be mutually composed to δ_P.

States of APLC machine are determined by its storage S. If $b \in S$ is any boolean variable then its value can be always observed as an assignment $b := [\![e]\!]\, S$, where e is an expression (evaluated with respect to a momentary state of storage S) that embodies some dependency of b value on another variables and equals to b variable itself in the simplest case i.e., $b := [\![b]\!]\, S$ (see also page 36).

Any state of APLC machine is transfered to subsequent one during an APLC instruction step. The step assigns new values to some subset of S variables and therefore each APLC instruction can be represented by one or more assignments, which will be called *t-assignments* (transfer assignments) in this thesis.

Let the evaluation method for a set of t-assignments be: "*We first evaluate the expressions of all t-assignments and store their results in temporary variables. Finally, we assign the temporary variables to variables specified on left sides of the t-assignments.*"

For instance, the instructions of APLC program "REdge x; Set z;" performing the operations: "set z to 1 on the rising edge of x", are equivalent to two sets of t-assignments:

$$\begin{aligned} X_{REdge\_x} &= \{(f_{reg} := [\![rFreg \wedge \neg x]\!]\, S), (x := [\![f_{reg}]\!]\, S)\} \\ X_{Set\_z} &= \{(z := [\![f_{reg} \vee z]\!]\, S)\} \end{aligned}$$

where $[\![\ ]\!]\, S$ only emphasizes an evaluation based on a momentary content of APLC storage S.

The evaluation method above, which will be precisely defined in the next subsection, gives correct results not only for X_{REdge_x} and X_{Set_z} but also for more complex APLC instructions as FSwp, which is describable by the t-assignments: [9]

$$X_{FSwp} = \{(f_{reg} := [\![e_1]\!]\, S), (e_1 := [\![f_{reg}]\!]\, S)\}$$

Employing the conversion we can express very complex operations as some sets of t-assignments. The main question considered in this chapter is how to compose them.

If X_{REdge_x} and X_{Set_z} are composed by some 'tricky' substitution to $X_{Scan} = X_{Set\_z} \circ X_{REdge\_x}$ we obtain t-assignments for the both instructions,

[9] X_{FSwp} leads to the program that is if written in Pascal syntax: `tmp_freg:=e1; tmp_e1:=freg; freg:=tmp_freg; e1:=tmp_e1;` There is the redundant usage of `tmp_freg` variable, but the swap works correctly.

from which we calculate δ_P. Notice inverse order of the both sets compare to APLC program. [10]

Naive algorithm of X evaluation could look as follows.

Naive algorithm

1. Each APLC c instruction with its operand is expressed as some set of t-assignments T_c.

2. The sequence consisting of codes T_{c_i} executed in i APLC steps is created and t-assignments of APLC program scan are calculated as composition $T_{Scan} = T_{c_n} \circ T_{c_{n-1}} \circ \ldots \circ T_{c_1}$

3. Branching of conditional jumps and subroutines compose as
$$T_{af} \circ ((cond \wedge T_{true}) \vee ((\neg cond) \wedge T_{false})) \circ T_{bf}$$
where T_{bf} and T_{af} are t-assignments of operations before branching and after joining the both branches, and *cond* represents branching condition.

The algorithm outlined in [Šus02] works but if we consider worst case program in Table 3.4, which does not manipulate data at all, we see that naive algorithm will learn this fact after calculating $2^n - 1$ compositions where n represents the number of instructions.

Moreover, applying our naive algorithm to another worst case APLC program with nesting limit (Table 3.5 on page 48) leads to multiple evaluations. Last block compositions are calculated $\left(\frac{n}{k}\right)^{k-1}$ times, the compositions of the previous block $\left(\frac{n}{k}\right)^{k-2}$ times and so on.

Naive algorithm resembles ill recursive program, which repeatedly evaluates already evaluated expressions, as for example well known recursive definition of Fibonacci function [Gra96, pg. 116].

$$\begin{aligned} \mathrm{Fib}(0) &= \mathrm{Fib}(1) = 1 \\ \mathrm{Fib}(n) &= \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) \end{aligned}$$

To improve naive algorithm, some parts of APLC program should be precalculated and skillfully composed as prepared partial functions, but, unfortunately, ordinary composition does not generally belong among associative operations, as we will show in Lemma 3.6 on page 58.

Therefore we must create a proper composition with associative property. We begin this by introducing the representation of variable values as t-assignments and by outlining necessary definitions and properties used for associative transfer sets.

[10] Although defining $\circ$ composition operator in inverse order as $X_{REdge\_x} \circ X_{Set\_z}$ would better match APLC machine operation we do not change 'tradition' established in formal language literature and further below we define the composition according to established notation.

3.3.2 Weak Composition of T-sets

Definition 3.12 *Let $bexp \in Bexp^+$ be any expression generated by Gbexp grammar (see page 35) then the domain of bexp is defined:*

$$\mathrm{dom}(bexp) \overset{df}{=} \{b_i \in \mathcal{B} \mid b_i \textit{ is used in } bexp\}$$

Because $Bexp^+$ grammar does not contain ε (see page 35), *bexp* is always non-empty, but its domain will be empty if *bexp* is a constant.

Definition 3.13 *Let $b \in S$ by any boolean variable from a finite non-empty storage $S \subset \mathcal{B}$, $bexp \in Bexp^+$ be any expression satisfying* $\mathrm{dom}(bexp) \subset S$, *and $b := [\![bexp]\!]\,S$ be the assignment operation of bexp (evaluated with respect to S) to b variable. We define*

$$\begin{array}{rrcl} \text{t-assignment:} & \hat{b}[\![bexp]\!] & \overset{df}{=} & b := [\![bexp]\!]\,S \\ \text{domain of t-assignment:} & \mathrm{dom}(\hat{b}[\![bexp]\!]) & \overset{df}{=} & \mathrm{dom}(bexp) \\ \text{codomain of t-assignment:} & \mathrm{co}(\hat{b}[\![bexp]\!]) & \overset{df}{=} & b \end{array}$$

T-assignment $\hat{b}[\![bexp]\!]$ is called canonical t-assignment *if $bexp \equiv b$. We denote the set of all t-assignments for S variables by $\widehat{\mathcal{B}}(S)$.*

To manipulate with t-assignments without cumulating too many symbols, we have used special notation for variables and their t-assignments. We have denoted the t-assignment by the same label as the variable, which it belongs to, but with hat accent, i.e., x variable has $\hat{x}$ t-assignment. To simplify orientation in the following paragraphs, we will also hat-accent all further objects related to t-assignments.

Any t-assignment $\hat{x}[\![bexp]\!] \in \widehat{\mathcal{B}}(S)$ can be expressed by several ways according to our momentary assumption about its structure. To increase readability of following paragraphs, we present the main variants:

- $\hat{x}[\![bexp]\!]$ - the t-assignment for x variable with *bexp* expression,
- $\hat{x}[\![x]\!]$ - the canonical t-assignment for x variable, i.e., its the expression equals to x variable itself, and
- $\hat{x}$ - any t-assignment for x variable with an arbitrary $bexp \in Bexp^+$.

T-assignments will be primed or subscribed. The symbols $\hat{x_i}$, $\hat{x_j}$, and $\hat{y}$ represent t-assignments for three (possibly different) variables x_i, x_j, and y. If we need to distinguish among several t-assignments for one identical variable, we will always write them in their full forms — symbols $\hat{x}[\![bexp_1]\!]$ and $\hat{x}[\![bexp_2]\!]$ stand for two (possibly different) t-assignments for one x variable.

The equality of t-assignments is determined by belonging to the same variable and their equivalent expressions in the meaning of boolean equivalence.

Definition 3.14 *Let $\hat{x}[\![bexp_x]\!] \in \widehat{\mathcal{B}}$ and $\hat{y}[\![bexp_y]\!] \in \widehat{\mathcal{B}}$ be two t-assignments. Binary relation $\hat{x} \mathrel{\widehat{=}} \hat{y}$ is defined as concurrent satisfaction of two following conditions:* $\mathrm{co}(\hat{x}) = \mathrm{co}(\hat{y})$ *and* $bexp_x \equiv bexp_y$.

If $\widehat{=}$ relation is not satisfied for some t-assignments $\hat{x}, \hat{y} \in \widehat{\mathcal{B}}$ then we will emphasize this fact by the negated symbol $\hat{x} \mathrel{\widehat{\neq}} \hat{y}$.

Lemma 3.4 *Binary relation $\widehat{=}$ on set $\widehat{\mathcal{B}}$ is equivalence relation.*

Proof: The relation is certainly reflexive and symmetric due to comparing operands by $=$ so we prove only its transitivity.

Let $\hat{x}[\![bexp_x]\!], \hat{y}[\![bexp_y]\!], \hat{z}[\![bexp_z]\!] \in \widehat{\mathcal{B}}$ be three t-assignments, which satisfy $\hat{x} \mathrel{\widehat{=}} \hat{y}$ and $\hat{y} \mathrel{\widehat{=}} \hat{z}$. Applying Definition 3.14 we obtain four equations

$$\begin{array}{rcl} \mathrm{co}(\hat{x}) & = & \mathrm{co}(\hat{y}) \\ bexp_x & \equiv & bexp_y \\ \mathrm{co}(\hat{y}) & = & \mathrm{co}(\hat{z}) \\ bexp_y & \equiv & bexp_z \end{array}$$

from which directly follows that $\hat{x} \mathrel{\widehat{=}} \hat{z}$. □

Since $\widehat{=}$ is an equivalence relation, the decomposition into classes of the equivalence on $\widehat{\mathcal{B}}$ exists (Proposition A.2 on page 138). If $\hat{x}[\![bexp]\!]$ is a t-assignment, then whole equivalence class

$$\widetilde{R}(\hat{x}[\![bexp]\!]) \overset{df}{=} \left\{ \hat{x}[\![bexp_i]\!] \in \widehat{\mathcal{B}}(S) \mid \hat{x}[\![bexp_i]\!] \mathrel{\widehat{=}} \hat{x}[\![bexp]\!] \right\} \tag{3.6}$$

can be considered as one element written in several different forms but always with equal value. Because decomposition splits $\widehat{\mathcal{B}}$ into disjoint subsets (see Proposition A.2), each subset is a unique component in terms of some operations properly defined, what we will provide in all further definitions.

The composition of more t-assignments at once requires the definition of the transfer sets to specify required replacements.

Definition 3.15 (Transfer Set) *A subset $\widehat{X} \subseteq \widehat{\mathcal{B}}(S)$ is called a* transfer set *on S, if $\widehat{X}$ satisfies for all $\hat{x_i}, \hat{x_j} \in X$ that $\mathrm{co}(\hat{x_i}) = \mathrm{co}(\hat{x_j})$ implies $i = j$. We denote the set of all transfer sets for S variables by $\widehat{\mathcal{S}}(S)$, i.e., $\widehat{X} \in \widehat{\mathcal{S}}(S)$.*

In other words, any transfer set contains at most one transfer function for each variable in S.

Example 3.4

Let $S = \{x, y, z\}$ be a PLC storage then $\widehat{X} = \{\hat{x}, \hat{y}\}$ is a transfer set on S, i.e., $\widehat{X} \in \widehat{\mathcal{S}}(S)$, but $Y = \{\hat{x}[\![y \vee z]\!], \hat{x}[\![y \wedge z]\!]\}$ is never a transfer set, because Y contains two t-assignments for x, and $\widehat{Z} = \{\hat{x}, \hat{a}\}$ is not a transfer sets on S, $\widehat{Z} \notin \widehat{\mathcal{S}}(S)$, because $\mathrm{co}(\hat{a}) \notin S$.

Manipulation with transfer set requires testing the presence of a transfer function for given variable $x \in S$.

Definition 3.16 *Binary relation $\widehat{\in}$ on sets S and $\widehat{\mathcal{B}}(S)$ is defined for all $\widehat{X} \in \widehat{\mathcal{S}}(S)$ and $x \in S$ as*

$$\widehat{\in} \overset{df}{=} \begin{array}{ll} x \mathbin{\widehat{\in}} \widehat{X} & \textit{iff } \exists\, \hat{x}[\![bexp]\!] \in \widehat{X} \textit{ such that } x = \mathrm{co}(\hat{x}[\![bexp]\!]) \\ x \mathbin{\widehat{\notin}} \widehat{X} & \textit{otherwise.} \end{array}$$

The composition of transfer sets is based on the *concurrent substitution* that was outlined on page 52. We define it here as mapping from variables in S to terms of *Gbexp* grammar. [11]

Definition 3.17 *Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be a transfer set and $bexp_{dest} \in Bexp^+$ be any expression.* Concurrent substitution *$\widehat{X} \rightsquigarrow bexp_{dest}$ is defined as such operation whose result is logically equivalent to the expression obtained by these consecutive steps:*

1. *For all $\hat{x}_i[\![bexp_i]\!] \in \widehat{X}$:*
   while $x_i \in \mathrm{dom}(bexp_{dest})$ (where $x_i = \mathrm{co}(\hat{x}_i[\![bexp_i]\!])$), x_i occurrence in $bexp_{dest}$ is replaced by some not interchangeable reference to x_i.
2. *For all $\hat{x}_i[\![bexp_i]\!] \in \widehat{X}$:*
   while the result of the previous step (modified expression $bexp_{dest}$) contains a reference to $x_i = \mathrm{co}(\hat{x}_i[\![bexp_i]\!])$ then x_i reference is replaced by "$(bexp_i)$" i.e., the expression of $\hat{x}_i[\![bexp_i]\!]$ enclosed inside parentheses.

The main purpose of the definition above is to exclude cyclic substitutions without detailed reasoning about an algorithm for this operation.

Example 3.5

Given concurrent substitution: $\{\hat{x}[\![x \wedge y]\!], \hat{y}[\![\neg x \wedge \neg y]\!]\} \rightsquigarrow \hat{c}[\![(x \vee y) \wedge x]\!]$
Direct application of the first step described in the definition above yields

$$\hat{c}[\![(\underline{\hat{x}} \vee \underline{\hat{y}}) \wedge \underline{\hat{x}}]\!]$$

[11] When the substitution is applied to a term, all occurrences of variables which appear in both the term and the domain of the substitution are replaced by their images under the substitution mapping [Fit90]

where underlining emphasizes that we have replaced variables by some unique references to the t-assignments that are not be their identifiers. The second step yields

$$\hat{c}[\![((x \wedge y) \vee (\neg x \wedge \neg y)) \wedge (x \wedge y)]\!]$$

but another acceptable results are also

$$\hat{c}[\![(x \equiv y) \wedge (x \wedge y)]\!] \quad \text{or} \quad \hat{c}[\![x \wedge y]\!]$$

because all expressions in three last t-assignments are logically equivalent.

Definition 3.18 (Weak composition) *Weak composition $\widehat{Z} = \widehat{X} \circ \widehat{Y}$ of two given transfer sets $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ is the transfer set $\widehat{Z} \in \widehat{\mathcal{S}}(S)$, $|\widehat{Z}| = |\widehat{X}|$ with t-assignments $\hat{x}_i[\![bexp_{z,i}]\!] \in \widehat{Z}$, $i \in I, |I| = |\widehat{X}|$, constructed by the following algorithm:*

$$\hat{x}_i[\![bexp_{z,i}]\!] \quad = \quad \hat{x}_i\left[\!\!\left[\widehat{Y} \rightsquigarrow bexp_{x,i}\right]\!\!\right] \text{ where } \hat{x}_i[\![bexp_{x,i}]\!] \in \widehat{X} \tag{3.7}$$

Example 3.6

Let $S = \{t, u, x, y, z\}$ be a set of variables. The compositional operation of two transfer sets is given as:

$$\widehat{Z} = \widehat{X} \circ \widehat{Y} = \{\hat{x}[\![x \vee y]\!], \hat{y}[\![x \wedge y]\!]\} \;\circ\; \{\hat{y}[\![x \wedge z]\!]\}$$

which leads to the simple substitution of y variables in $\widehat{X}$ by $(x \wedge z)$

$$\widehat{Z} = \{\hat{x}[\![x \vee (x \wedge z)]\!], \hat{y}[\![x \wedge (x \wedge z)]\!]\} \quad .$$

We obtain after simplifying:

$$\widehat{Z} = \{\hat{x}[\![x]\!], \hat{y}[\![x \wedge z]\!]\} \quad .$$

If we have two transfer sets and their composition given as $\widehat{X} \circ \widehat{Y} = \{\hat{x}[\![bexp_x]\!]\} \circ \{\hat{y}[\![bexp_y]\!]\}$ and variable $y \in \mathrm{dom}(\hat{x}[\![bexp_x]\!])$, then the result is obtained by substituting $(bexp_y)$ instead of y variable in $bexp_x$. Otherwise, if $y \notin \mathrm{dom}(\hat{x}[\![bexp_x]\!])$ then the result is unchanged $\widehat{X}$, because nothing is substituted.

Lemma 3.5 *Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be any transfer set on S then it holds that:*

$$\widehat{X} = \widehat{X} \circ \emptyset \quad \text{and} \quad \emptyset = \emptyset \circ \widehat{X} \tag{3.8}$$

Proof: The lemma follows directly from Definition 3.18. If one operand is the empty set then $\circ$ always returns its left operand. $\square$

In case of composing $\widehat{X} \circ \widehat{Y}$, where $|\widehat{Y}| = 1$, the operation $\circ$ corresponds to a classic substitution, but that will not hold for an associative composition of transfer sets, which we aim to, because the weak composition shows one troublesome property.

Lemma 3.6 *The weak composition $\circ$ is not associative on $\widehat{\mathcal{S}}(S)$.*

Proof: We prove the lemma by the example. Let $S = \{x, y\}$ be storage and $\widehat{X} = \{\hat{x}[\![x \wedge y]\!]\}$, $\widehat{Y} = \{\hat{y}[\![x]\!]\}$, and $\widehat{Z} = \{\hat{x}[\![\neg x]\!]\}$ three transfer sets on S then the composition:

$$\begin{aligned}(\widehat{X} \circ \widehat{Y}) \circ \widehat{Z} &= (\ \{\ \hat{x}[\![x \wedge y]\!]\} \circ \{\ \hat{y}[\![x]\!]\ \}\ ) \circ \{\ \hat{x}[\![\neg x]\!]\ \} \\ &= \{\ \hat{x}[\![x \wedge x]\!]\ \} \circ \{\ \hat{x}[\![\neg x]\!]\ \} = \{\ \hat{x}[\![\neg x]\!]\ \}\end{aligned} \tag{3.9}$$

If we first compose the two last sets:

$$\begin{aligned}\widehat{X} \circ (\widehat{Y} \circ \widehat{Z}) &= \{\ \hat{x}[\![x \wedge y]\!]\} \circ (\ \{\ \hat{y}[\![x]\!]\ \} \circ \{\ \hat{x}[\![\neg x]\!]\ \}\ ) \\ &= \{\ \hat{x}[\![x \wedge y]\!]\ \} \circ \{\ \hat{y}[\![\neg x]\!]\ \} \\ &= \{\ \hat{x}[\![x \wedge \neg x]\!]\ \} = \{\ \hat{x}[\![0]\!]\ \}\end{aligned} \tag{3.10}$$

□

The non-associative behavior has appeared in the example due to different variables affected by transfer sets: $x \mathrel{\widehat{\notin}} \widehat{Y}$, but $x \mathrel{\widehat{\in}} \widehat{X}$ and $x \mathrel{\widehat{\in}} \widehat{Z}$, therefore there was the absorption of $\hat{x}[\![\neg x]\!]$ in $\widehat{Y} \circ \widehat{Z}$ composition and the result did not contain x t-assignment — we have lost a part of information.

The restriction of $\widehat{\mathcal{S}}(S)$ to transfer sets for one given variable $x \in S$ offers the simplest solution. This subset, denoted by $\widehat{\mathcal{S}}(S/x) \subset \widehat{\mathcal{S}}(S)$, is defined as

$$\widehat{\mathcal{S}}(S/x) \overset{df}{=} \left\{ \widehat{X_i} \in \widehat{\mathcal{S}}(S)\ \mid x_i \mathrel{\widehat{\notin}} \widehat{X_i} \text{ for all } x_i \neq x, x_i \in \mathcal{B} \right\} \tag{3.11}$$

Now more hopeful proposition holds.

Proposition 3.3 *The operation $\circ$ is associative on $\widehat{\mathcal{S}}(S/x)$ for non-empty transfer sets.*

Proof: We should prove Equation 3.12 for $\widehat{X}_1, \widehat{X}_2, \widehat{X}_3 \in \widehat{\mathcal{S}}(S/x)$

$$\widehat{X}_1 \circ \left(\widehat{X}_2 \circ \widehat{X}_3 \right) \overset{?}{=} \left(\widehat{X}_1 \circ \widehat{X}_2 \right) \circ \widehat{X}_3 \tag{3.12}$$

First, we prove the necessity of assuming non-empty sets by the following example. Let X_2 be an empty transfer set then Lemma 3.5 yields:

$$\begin{aligned}\widehat{X}_1 \circ \left(\emptyset \circ \widehat{X}_3 \right) &\overset{?}{=} \left(\widehat{X}_1 \circ \emptyset \right) \circ \widehat{X}_3 \\ \widehat{X}_1 \circ \emptyset = \widehat{X}_1 &\neq \widehat{X}_1 \circ \widehat{X}_3\end{aligned}$$

therefore the proposition must assume non-empty transfer sets.

According to Definition 3.15 any transfer set contains at most one t-assignment for each variable in S, from which follows that $|\widehat{X}_i| = 1$ for any non-empty transfer set $\emptyset \neq \widehat{X}_i \in \widehat{\mathcal{S}}(S/x)$. If all $\widehat{X}_i$ are non-empty Equation 3.12 can write them as:

$$\begin{aligned} & \{\ \hat{x}[\![\mathit{bexp}_1]\!]\ \} \circ (\ \{\ \hat{x}[\![\mathit{bexp}_2]\!]\ \} \circ \{\ \hat{x}[\![\mathit{bexp}_3]\!]\ \}\ ) \\ \overset{?}{=}\ & (\ \{\ \hat{x}[\![\mathit{bexp}_1]\!]\ \} \circ \{\ \hat{x}[\![\mathit{bexp}_2]\!]\ \}\ ) \circ \{\ \hat{x}[\![\mathit{bexp}_3]\!]\ \} \end{aligned} \tag{3.13}$$

We proceed by considering presence of x in domains of $\hat{x}[\![\mathit{bexp}_1]\!]$ and $\hat{x}[\![\mathit{bexp}_2]\!]$. The both cases of $x \in \mathrm{dom}(\hat{x}[\![\mathit{bexp}_3]\!])$ or $x \notin \mathrm{dom}(\hat{x}[\![\mathit{bexp}_3]\!])$ are irrelevant because nothing is substituted into bexp_3.

First suppose $x \notin \mathrm{dom}(\hat{x}[\![\mathit{bexp}_1]\!])$. In such case, $\{\ \hat{x}[\![\mathit{bexp}_1]\!]\ \} \circ \widehat{X}_i$ yields $\{\ \hat{x}[\![\mathit{bexp}_1]\!]\ \}$ for any $\widehat{X}_i \in \widehat{\mathcal{S}}(S/x)$. Therefore the both sides of Equation 3.13 evaluate to $\{\ \hat{x}[\![\mathit{bexp}_1]\!]\ \}$ and associative law holds.

Now consider that $x \in \mathrm{dom}(\hat{x}[\![\mathit{bexp}_1]\!])$ and $x \notin \mathrm{dom}(\hat{x}[\![\mathit{bexp}_2]\!])$. The left hand side of Equation 3.13 yields $\{\ \hat{x}[\![\mathit{bexp}_1]\!]\ \} \circ \{\ \hat{x}[\![\mathit{bexp}_2]\!]\ \}$ because $\hat{x}[\![\mathit{bexp}_2]\!]$ does not contain x and so $\hat{x}[\![\mathit{bexp}_3]\!]$ is not substituted. In this proof only, let us write $\widehat{X}_{12} = \{\ \hat{x}[\![\mathit{bexp}_1]\!]\ \} \circ \{\ \hat{x}[\![\mathit{bexp}_2]\!]\ \}$.

In contrast, the right hand side of the equation evaluates $\widehat{X}_{12}$ first but its domain of does not contain x because all terms with x in bexp_1 have been replaced by bexp_2 without x variable since $x \notin \mathrm{dom}(\hat{x}[\![\mathit{bexp}_2]\!])$. Therefore $\widehat{X}_{12} \circ \{\ \hat{x}[\![\mathit{bexp}_3]\!]\ \}$ yields $\widehat{X}_{12}$ and associative law is also valid.

Now consider the case $x \in \mathrm{dom}(\hat{x}[\![\mathit{bexp}_1]\!])$ and $x \in \mathrm{dom}(\hat{x}[\![\mathit{bexp}_2]\!])$. Decomposing operator $\circ$ we get

$$\begin{aligned} & \hat{x}\,[\![\hat{x}[\![\mathit{bexp}_3]\!] \rightsquigarrow (\hat{x}[\![\mathit{bexp}_2]\!] \rightsquigarrow \mathit{bexp}_1)]\!] \\ =\ & \hat{x}\,[\![(\hat{x}[\![\mathit{bexp}_3]\!] \rightsquigarrow \hat{x}[\![\mathit{bexp}_2]\!]) \rightsquigarrow \mathit{bexp}_1]\!] \end{aligned} \tag{3.14}$$

Proving associativity of $\circ$ is transformed to verifying the same for $\rightsquigarrow$ but it holds because all substitutions are applied only to x.

Replacing all occurrences of x in bexp_1 by bexp_2 yields a result with terms that contain only x variables brought by bexp_2. These will be replaced by bexp_3 consecutively. Reversing procedure we obtain the same after first substituting bexp_3 instead of each x in bexp_2 and then using the result for replacing all x in bexp_1. □

Proposition 3.4 $\widehat{\mathbb{G}}(S/x) = (\widehat{\mathcal{S}}(S/x)/\hat{=}, \hat{=}, \circ)$ *is semigroup.*

Proof: We have already proved that $\circ$ operation is associative so we only show that $\widehat{\mathcal{S}}(S/x)$ is closed under the semigroup operation $\circ$ with respect to $\hat{=}$ equality. This evidently holds since the result of applying $\circ$ to a t-assignment of x variable is again a t-assignment of the same variable.

Language Bexp^+ is also closed for any $\rightsquigarrow$ concurrent substitution because its grammar (see page 35) generates all logical operation with the aid of non-terminal symbol *Gbexp*, which constitutes both any allowable expression or variable identifier. Therefore substituting some expression instead of a variable identifier yields again an expression belonging to the same language. $\square$

For non-empty elements of $\widehat{\mathcal{S}}(S/x)$, we define Boolean algebra that will be extended to $\widehat{\mathcal{S}}(S)$ in the next subsection. First, we present boolean operations.

Definition 3.19 *Let* $\{\ \hat{x}[\![bexp_1]\!]\ \}, \{\ \hat{x}[\![bexp_2]\!]\ \} \in \widehat{\mathcal{S}}(S/x)$ *be a non-empty transfer set. We define the following operations:*

$$\begin{aligned}
\{\ \hat{x}[\![bexp_1]\!]\ \} \odot \{\ \hat{x}[\![bexp_2]\!]\ \} &\overset{df}{=} \{\ \hat{x}[\![bexp_1 \odot bexp_2]\!]\ \} \\
\neg\{\ \hat{x}[\![bexp_1]\!]\ \} &\overset{df}{=} \hat{x}[\![\neg(bexp_1)]\!]
\end{aligned}$$

where $\odot$ *represents any boolean binary operation used in Gbexp grammar (see page 35).*

Notice enclosing operands in parentheses to solves any ambiguities caused by different precedences of boolean operations.

Proposition 3.5 *Lattice* $\widehat{\mathbb{A}}(S/x) \overset{df}{=} \Big(\ \widehat{\mathcal{S}}(S/x)/\widehat{=}, \wedge, \vee, \{\hat{x}[\![1]\!]\}, \{\hat{x}[\![0]\!]\}\ \Big)$ *is Boolean algebra.*

Proof: Members of the algebra are transfer sets with boolean expressions in t-assignments, for which all boolean laws will certainly hold.

The smallest and largest elements represents transfer sets $\{\hat{x}[\![0]\!]\}$ and $\{\hat{x}[\![1]\!]\}$ satisfying bounded below and above laws. The complement is given by $\neg\{\hat{x}\}$ operation defined in the definition.

Any transfer set can be certainly written in many forms, for example $\{\hat{x}[\![1]\!]\}$ and $\{\hat{x}\,[\![1+1]\!]\}$ are equal in terms of $\widehat{=}$ operation, but different in terms of $=$ operation. We have defined equivalence class $\widetilde{R}(\hat{x}[\![bexp]\!])$ on page 55. If we consider all members of similar equivalence class, its 'clumsy' definition (without using operations introduced latter) is:

$$\widetilde{R}(\widehat{X}/x) \overset{df}{=} \left\{ \{\hat{x}[\![bexp_i]\!]\} \in \widehat{\mathcal{S}}(S/x) \mid \begin{array}{c} \hat{x}[\![bexp_i]\!] \ \widehat{=}\ \hat{x}[\![bexp]\!]\,, \\ \text{where } \hat{x}[\![bexp]\!] \in \widehat{X} \end{array} \right\}$$

as one unique element of the algebra, as done in the definition, then the smallest and largest elements will be unique and each element will have only one complement as follows from algebra's axioms. $\square$

3.3.3 Monoid and Boolean Algebra of Transfer Sets

In this part, we will extend previous definitions to transfer sets, which were introduced on page 55. We will create associative compositions needed for improvement of our naive algorithm outlined on page 53).

We begin by the extensions of a transfer set that adds canonical t-assignments for all variables in S, whose t-assignments are missing in the transfer set (see page 54).

Definition 3.20 *Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be a transfer set on variable set S. The extension of $\widehat{X}$, denoted by $\widehat{X} \uparrow S$, is the set with cardinality $|\widehat{X} \uparrow S| = |S|$ whose members are $\hat{x}_i$ t-assignments defined for all $x_i \in S$ as*

$$\hat{x}_i \overset{df}{=} \begin{cases} \hat{x}_i & \textit{if } x_i \,\widehat{\in}\, \widehat{X} \textit{ and thus } \exists \hat{x}_i \in \widehat{X} \textit{ satisfying } \operatorname{co}(\hat{x}_i) = x_i \\ \hat{x}_i\,[\![x_i]\!] & \textit{if } x_i \,\widehat{\notin}\, \widehat{X} \end{cases}$$

The set $\emptyset \uparrow S$, called *canonical transfer set* of S contains canonical t-assignments of all variables in S and it has an exclusive position among all transfer sets. To emphasize its fundamental importance, let us denoted this set by $\widehat{\mathcal{E}}^S$.

Definition 3.21 *Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be a transfer set on variable set S. The compression of $\widehat{X}$, denoted by $\widehat{X}\downarrow$, is defined as*

$$\widehat{X}\downarrow \overset{df}{=} \left\{ \hat{x}_i[\![bexp]\!] \in \widehat{X} \mid \hat{x}_i[\![bexp]\!] \,\widehat{\not\equiv}\, \hat{x}_i[\![x_i]\!] \right\}$$
$$\textit{where } x_i = \operatorname{co}(\hat{x}_i) \ .$$

The compression expresses meaning of $\downarrow$ operation that serves for packing transfer sets but its primary purpose consists mainly in portability among different S. Operators $\uparrow$ and $\downarrow$ allow specifying only such t-assignments, which express required operations on some subset of variables $S_0 \subset S$ that includes performed changes in S. This transfer set S_0 can be extended to any subset S_i of variables such that $S_i \supset S_0$.

Example 3.7

Let

$$\widehat{T} = \left\{ \begin{array}{l} \hat{x}[\![x]\!]\,, \\ \hat{y}[\![x \vee y]\!] \end{array} \right\}$$

be a transfer set on $S = \{x, y, z\}$ then

$$\widehat{T}\uparrow S = \left\{ \begin{array}{l} \hat{x}[\![x]\!]\,, \\ \hat{y}[\![x \vee y]\!]\,, \\ \hat{z}[\![z]\!] \end{array} \right\} \quad \text{and} \quad \widehat{T}\downarrow = \{\hat{y}[\![x \vee y]\!]\}$$

The extension and compression are generally not inverse operations but they will be if we narrow $\widehat{\mathcal{S}}(S)$ to subsets invariable with respect to these operations.

$$\widehat{\mathcal{S}}(S)\downarrow \overset{df}{=} \left\{\widehat{X} \in \widehat{\mathcal{S}}(S) \mid \widehat{X} = \widehat{X}\downarrow\right\} \tag{3.15}$$

$$\widehat{\mathcal{S}}(S)\uparrow S \overset{df}{=} \left\{\widehat{X} \in \widehat{\mathcal{S}}(S) \mid \widehat{X} = \widehat{X}\uparrow S\right\} \tag{3.16}$$

Lemma 3.7 *Operators $\uparrow$ and $\downarrow$ are bijective mappings between $\widehat{\mathcal{S}}(S)\downarrow$ and $\widehat{\mathcal{S}}(S)\uparrow S$.*

Proof: The lemma directly follows from the definition. Let us consider transfer set $\widehat{X} \in \widehat{\mathcal{S}}(S)\downarrow$ then $\widehat{X}$ is invariable with respect to $\downarrow$ compression because it does not contain any canonical t-assignment — they all were removed by the compression.

Its $\uparrow$ extension will add only canonical t-assignments so their removing yields $\widehat{X}$ again, because adding and removing canonical t-assignment do not change remaining t-assignments. Therefore, each transfer set is also mapped to unique element and

$$\mathrm{fn}: \widehat{\mathcal{S}}(S)\downarrow \leftrightarrow \widehat{\mathcal{S}}(S)\uparrow S$$

□

Furthermore, we extend the equivalence relation of t-assignments introduced in Definition 3.14 for transfer sets.

Definition 3.22 *Let $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ be two transfer sets on S then $\widehat{X} \mathrel{\widehat{=}} \widehat{Y}$ if, for all $\hat{x}_i[\![bexp_1]\!] \in (\ \widehat{X} \uparrow S\ )$, $\hat{x}_i[\![bexp_2]\!] \in (\ \widehat{Y} \uparrow S\ )$ exists such that $\hat{x}_i[\![bexp_1]\!] \mathrel{\widehat{=}} \hat{x}_i[\![bexp_2]\!]$.*

Every transfer set contains at most one t-assignment for each variable (Definition 3.15) so two transfer sets $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ will be equal when one of the following conditions is satisfied for any given variable $x \in S$

1. If $x \mathrel{\widehat{\in}} \widehat{X}$ and $x \mathrel{\widehat{\in}} \widehat{Y}$ then both the t-assignments of x have their expressions logically equal.
2. If x has a t-assignment only in one of the both sets, then this t-assignment is $\widehat{=}$ equivalent to canonical t-assignment.
3. x has no t-assignment in the both transfer sets.

Considerations above support following proposition.

Proposition 3.6 *Binary relation $\widehat{=}$ on set $\widehat{\mathcal{S}}(S)$ is equivalence.*

Proof: The relation is certainly reflexive and symmetric because elements are compared with the aid of $=$ and $\equiv$ relations and they are extended on S before their comparison so we must prove only transitivity.

Let $\widehat{X}, \widehat{Y}, \widehat{Z} \in \widehat{\mathcal{S}}(S)$ be three transfer sets. Transitivity law for $\widehat{=}$ is $\widehat{X} \mathrel{\widehat{=}} \widehat{Y}$ and $\widehat{Y} \mathrel{\widehat{=}} \widehat{Z}$ imply $\widehat{X} \mathrel{\widehat{=}} \widehat{Z}$. This will hold only if t-assignments belonging to any selected variable satisfy the same law because only such t-assignments are mutually compared.

Let $b \in S$ be a variable and let us write $\hat{b}[\![\mathit{bexp}_X]\!], \hat{b}[\![\mathit{bexp}_Y]\!], \hat{b}[\![\mathit{bexp}_Z]\!]$ for b t-assignments in extended transfer sets $\widehat{X} \uparrow S$, $\widehat{Y} \uparrow S$, $\widehat{Z} \uparrow S$. These t-assignments always exist according to Definition 3.20, so we rewrite the transitivity law as:

$$\begin{aligned} & \hat{b}[\![\mathit{bexp}_X]\!] \mathrel{\widehat{=}} \hat{b}[\![\mathit{bexp}_Y]\!] \;\wedge\; \hat{b}[\![\mathit{bexp}_Y]\!] \mathrel{\widehat{=}} \hat{b}[\![\mathit{bexp}_Z]\!] \\ \Rightarrow\; & \hat{b}[\![\mathit{bexp}_X]\!] \mathrel{\widehat{=}} \hat{b}[\![\mathit{bexp}_Z]\!] \end{aligned} \tag{3.17}$$

We have already proved that transitivity holds for three arbitrary t-assignment of one variable in Lemma 3.4 on page 55.

Now we must take in account non existent t-assignment in some transfer set. If it happens, extension operator $\uparrow$ supplies canonical t-assignment instead of missing t-assignment. In such case remaining t-assignments must be also equivalent to canonical t-assignment, otherwise Statement 3.17 could not hold. Therefore, transitivity is valid for all cases. $\square$

Similar consideration as for Definition 3.14 can be also given for transfer sets. If $\widehat{X} \in \widehat{\mathcal{S}}(S)$ is a transfer set then whole equivalence class

$$\widetilde{R}(\widehat{X}) \overset{df}{=} \left\{ \widehat{X}_i \in \widehat{\mathcal{S}}(S) \mid \widehat{X}_i \mathrel{\widehat{=}} \widehat{X} \right\} \tag{3.18}$$

represents one element written in several different forms but always with the same value in terms of some operations properly defined.

Definition 3.23 *Let $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ be two transfer sets on S with extensions $\widehat{X}' = \widehat{X} \uparrow S$ and $\widehat{Y}' = \widehat{Y} \uparrow S$. We define the following operations:*

$$\begin{aligned} \widehat{X} \odot \widehat{Y} &\overset{df}{=} \left\{ \hat{x}_i[\![\mathit{bexp}_X \odot \mathit{bexp}_Y]\!] \mid \hat{x}_i[\![\mathit{bexp}_X]\!] \in \widehat{X}' \text{ and } \hat{x}_i[\![\mathit{bexp}_Y]\!] \in \widehat{Y}' \right\} \downarrow \\ \neg \widehat{X} &\overset{df}{=} \left\{ \hat{x}_i[\![\neg \mathit{bexp}_X]\!] \mid \hat{x}_i[\![\neg \mathit{bexp}_X]\!] \in \widehat{X}' \right\} \downarrow \end{aligned}$$

where $\odot$ represents any boolean binary operation used in Gbexp grammar (see page 35).

Let us write $\widehat{\Theta}(\mathit{bexp})^S$ for the transfer set defined as

$$\widehat{\Theta}(\mathit{bexp})^S \overset{df}{=} \{\hat{x}_i\, [\![\mathit{bexp}]\!] \mid \text{ for all } x \in S,\ x = \mathrm{co}(\hat{x}_i)\} \tag{3.19}$$

where *bexp* $\in$ *Bexp*$^+$ is an arbitrary expression, which is assigned to all t-assignments. It allows defining Boolean algebra of transfer sets.

Proposition 3.7 *Lattice* $\widehat{\mathbb{A}}(S) \overset{df}{=} (\widehat{\mathcal{S}}(S)/\widehat{=}, \widehat{=}, \wedge, \vee, \widehat{\Theta}(1)^S, \widehat{\Theta}(0)^S)$ *is Boolean algebra.*

Proof: The lattice $\widehat{\mathbb{A}}(S/x) = \Big(\widehat{\mathcal{S}}(S/x)/\widehat{=}, \wedge, \vee, \{\hat{x}[\![1]\!]\}, \{\hat{x}[\![0]\!]\} \Big)$ is Boolean algebra for $\widehat{\mathcal{S}}(S/x)$, which was proved in Proposition 3.5 on 60, for transfer sets with t-assignments for one variable $x \in S$. Because boolean operations with transfer sets are done as the union of operations with t-assignments for each S variable, the same surely satisfy transfer sets. Whole $\widehat{\mathcal{S}}(S)$ is closed under algebra operations since all $\widehat{\mathcal{S}}(S/x)$ are closed for all $x \in S$.

The smallest and largest elements represent $\widehat{\Theta}(1)^S$ and $\widehat{\Theta}(0)^S$ satisfying bounded below and above laws. The complement is given by $\neg\widehat{X}$. These elements are unique in terms of $\widehat{=}$ which was already discussed in the proof of Proposition 3.5. □

Example 3.8

Let

$$\widehat{T} = \{\hat{y}[\![z \vee y]\!]\} \quad \text{and} \quad \widehat{U} = \left\{ \begin{array}{l} \hat{y}[\![\neg z]\!]\,, \\ \hat{z}[\![x \wedge y]\!] \end{array} \right\}$$

be two transfer set on $S = \{x, y, z\}$ then:

$$\widehat{T} \wedge \widehat{U} \;=\; \left\{ \begin{array}{l} \hat{y}[\![(z \vee y) \wedge \neg z]\!]\,, \\ \hat{z}[\![z \wedge (x \wedge y)]\!] \end{array} \right\} \quad \text{and} \quad \neg\widehat{T} \;=\; \left\{ \begin{array}{l} \hat{x}[\![\neg x]\!]\,, \\ \hat{y}[\![\neg(x \vee y)]\!]\,, \\ \hat{z}[\![\neg z]\!] \end{array} \right\}$$

$\neg\widehat{T}$ contains t-assignments that are not in $\widehat{T}$, because $\widehat{T}$ was first extended by $\uparrow$ operator, which has added canonical t-assignments $\hat{x}[\![x]\!]$ and $\hat{z}[\![z]\!]$, and then $\widehat{T}\!\uparrow$ was negated.

In contrast, notice missing t-assignment for x variable in $\widehat{T} \wedge \widehat{U}$, because it equaled to $\hat{x}[\![x \wedge x]\!]$ after $\wedge$ operation and therefore it was removed by $\downarrow$ compression as $\widehat{=}$ equivalent of the canonical t-assignment $\hat{x}[\![x]\!]$.

The main operator for transfer sets is $\circledcirc$ composition operator.

Definition 3.24 (Composition) *Let* $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ *be two transfer sets on* S *then their* $\circledcirc$ *composition is defined as:*

$$\widehat{X} \circledcirc \widehat{Y} \;\overset{df}{=}\; \Big\{ \hat{x} \circ \widehat{Y} \mid \hat{x} \in \Big(\widehat{X}\!\uparrow S \Big) \Big\}\!\downarrow$$

Example 3.9

Let $\widehat{T}, \widehat{U} \in \widehat{\mathcal{S}}(S)$ be two transfer set on $S = \{x, y, z\}$ from the previous example. Their composition $\widehat{T} \circledcirc \widehat{U}$ can be written as

$$\widehat{T} \circledcirc \widehat{U} \quad = \quad \left((\{\hat{y}[\![z \vee y]\!]\} \uparrow S) \circledcirc \left\{ \begin{array}{l} \hat{y}[\![\neg z]\!]\,, \\ \hat{z}[\![x \wedge y]\!] \end{array} \right\} \right) \downarrow$$

Definition 3.20 and inner $\uparrow$ yields

$$\widehat{T} \circledcirc \widehat{U} \quad = \quad \left(\left\{ \begin{array}{l} \hat{x}[\![x]\!]\,, \\ \hat{y}[\![z \vee y]\!]\,, \\ \hat{z}[\![z]\!] \end{array} \right\} \circ \left\{ \begin{array}{l} \hat{y}[\![\neg z]\!]\,, \\ \hat{z}[\![x \wedge y]\!] \end{array} \right\} \right) \downarrow$$

expanding $\circ$ weak composition gives:

$$\widehat{T} \circledcirc \widehat{U} \quad = \quad \left\{ \begin{array}{l} \hat{x}[\![x]\!]\,, \\ \hat{y}[\![(x \wedge y) \vee \neg z]\!]\,, \\ \hat{z}[\![x \wedge y]\!] \end{array} \right\} \downarrow \quad = \quad \left\{ \begin{array}{l} \hat{y}[\![(x \wedge y) \vee \neg z]\!]\,, \\ \hat{z}[\![x \wedge y]\!] \end{array} \right\}$$

Lemma 3.8 *Let $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ be any transfer sets on S then:*

$$\begin{aligned} \widehat{X} \circledcirc \widehat{Y} \;&\widehat{=}\; \left(\widehat{X} \uparrow S \right) \circledcirc \widehat{Y} && (3.20) \\ &\widehat{=}\; \widehat{X} \circledcirc \left(\widehat{Y} \uparrow S \right) && (3.21) \\ &\widehat{=}\; \left(\widehat{X} \uparrow S \right) \circledcirc \left(\widehat{Y} \uparrow S \right) && (3.22) \\ &\widehat{=}\; \left(\widehat{X} \downarrow \right) \circledcirc \widehat{Y} && (3.23) \\ &\widehat{=}\; \widehat{X} \circledcirc \left(\widehat{Y} \downarrow \right) && (3.24) \\ &\widehat{=}\; \left(\widehat{X} \downarrow \right) \circledcirc \left(\widehat{Y} \downarrow \right) && (3.25) \end{aligned}$$

Proof: Because the extension is always applied to $\widehat{X}$ before the composition Equation 3.20 follows directly from Definition 3.24.

According to Definition 3.20, $\uparrow$ extension only add canonical t-assignments that will replace corresponding variables by themselves enclosed in parentheses when the concurrent substitution is performed (see Definition 3.17 on page 56), i.e., if $x \in S$ is any variable in a boolean expression then x is replaced by (x) when substituting canonical t-assingment $\hat{x}[\![x]\!]$. These modifications do not change logical value of boolean expressions, therefore the result does not depend of possible $\widehat{Y}$ extension and Equation 3.21 always holds.

Applying Equations 3.20 and 3.21 to $\widehat{X} \circledcirc \widehat{Y}$ we obtain Equation 3.22. Equations from 3.23 to 3.25 follows from bijective properties of $\uparrow$ and $\downarrow$ (Lemma 3.7). $\square$

Now we present the main theorem of this thesis for which validity we have created transfer sets.

Proposition 3.8 *The composition $\circledcirc$ is the associative operation on $\widehat{\mathcal{S}}(S)$.*

Proof: Utilizing Lemma 3.8 we take in account only transfer sets in $\widehat{\mathcal{S}}(S)\uparrow S$. We have proved in Lemma 3.7 that $\uparrow$ and $\downarrow$ operators perform mutual bijective mapping therefore outer compression operator does not change validity of results.

Let $\widehat{X}, \widehat{Y}, \widehat{Z} \in \widehat{\mathcal{S}}(S)\uparrow S$ be three extended transfer sets. In this case $\circledcirc$ associativity means

$$(\widehat{X} \circledcirc \widehat{Y}) \circledcirc \widehat{Z} = \widehat{X} \circledcirc (\widehat{Y} \circledcirc \widehat{Z}) \tag{3.26}$$

Equation 3.26 will hold if the following equation

$$(\hat{x} \circ \widehat{Y}) \circledcirc \widehat{Z} = \hat{x} \circ (\widehat{Y} \circledcirc \widehat{Z}) \tag{3.27}$$

is satisfied for all $\hat{x} \in \widehat{X}$ (Definition 3.24). We expand $\circ$ in front of $\hat{x}$ according Definition 3.18 on page 57

$$(\widehat{Y} \rightsquigarrow \hat{x}) \circledcirc \widehat{Z} = (\widehat{Y} \circledcirc \widehat{Z}) \rightsquigarrow \hat{x} \tag{3.28}$$

Now, we divide the proof into three parts. First, we will consider the flow of symbols in terms of languages to prove that the result contains only variables imported from a rightmost transfer set of Equation 3.26, then we will show that the associativity holds if no optimizations of boolean expressions are performed, and finally, we will reason about the optimizations.

We rewrite Equation 3.28 in terms of languages. Any t-assignment $\hat{x}_i \llbracket \mathit{bexp}_i \rrbracket$ contains an expression bexp_i that represents a string from language Bexp^+, which was generated by *Gbexp* grammar (defined on page 35).

E alphabet of language Bexp^+ consists of the symbols of S variables and literals, which will be denoted (only in this proof) by Λ. The literals include all symbols Bexp^+ alphabet that are different from variables i.e., all boolean operations allowed in *Gbexp* grammar, parentheses, and constants 0 and 1. Thus $\Lambda = E - S$.

In this proof only, we create marking of alphabet's symbols by superscripts. Alphabet S^x will represent S variable symbols with 'x' marks specifying that these symbols have their origin in $\widehat{X}$ expressions. Similarly, alphabets S^y and S^z belong to $\widehat{Y}$ and $\widehat{Z}$. S^x, S^y and S^z are entirely equal to S in terms of the grammar and manipulation with t-assignments. They only differ by auxiliary marks.

Using notation above and Definition A.13 we split the expression of $\hat{x}_i[\![bexp_i^x]\!] \in \widehat{X}$ string into the concatenation of n substrings

$$bexp_i^x = \lambda_{i,1}.s_{i,1}^x.\lambda_{i,2}.s_{i,2}^x\ \ldots\ .\lambda_{i,n_i-1}.s_{i,n_i-1}^x.\lambda_{i,n_i} \tag{3.29}$$

where $n_i > 0$ is an integer number, $s_{i,j}^x \in S^x$ are variable symbols and $\lambda_i \in \Lambda^+$ for $i = 2\ldots(n-1)$ are non-empty strings, whereas $\lambda_{i,1}, \lambda_{i,n_i} \in \Lambda^*$ are (possibly empty) strings. This follows directly from grammar's rules because all variable names must be separates by strings of Λ^+ language and any expression may begin or end by some string from Λ^* language.

Definition 3.17 describes the concurrent substitution as replacing each $s_{i,j}^x$ variable by the value of $\hat{s}_{i,j}^y \left[\!\left[bexp_{k,i}^y\right]\!\right]$ where co$\left(\hat{s}_{i,j}^y\right) = s_{i,j}^x$. Applying this to Equation 3.29 yields

$$bexp_i^{x\circledcirc y} = \lambda_{i,1}.\underline{(}.bexp_{k_1}^y.\underline{)}.\lambda_{i,2}.\underline{(}.bexp_{k_2}^y.\underline{)}.\ \ldots\ \lambda_{i,n_i-1}.\underline{(}.bexp_{k_{n_i-1}}^y.\underline{)}.\lambda_{i,n_i} \tag{3.30}$$

where symbols $\underline{(}, \underline{)} \in \Lambda$ specify ordinary parentheses which were only marked by underlines to distinguish them from mathematical parentheses, and terms $bexp_{k_j}^y$ represent some values of t-assignments $\hat{y}_{k_j} \in \widehat{Y}$ corresponding to original variables $s_{i,j}^x$. Indexes k_j form a list with (possibly) repeated values because one or more variables could occur more times in $bexp_i^x$.

If we consider properties of $bexp^{x\circledcirc y}$ we see that

$$\emptyset = \left\{s_i^x \in S^x \mid s_i^x \in bexp^{x\circledcirc y}\right\} \tag{3.31}$$

because all s_i^x have been replaced. String $bexp^{x\circledcirc y}$ contains only variables marked as belonging to $\widehat{Y}$ expressions. This follows directly from Definition 3.15 of transfer sets and $\uparrow$ extension operator (Definition 3.20).

Therefore, the result of composing $\hat{x}_i \circ \widehat{Y}$ is a string from the language defined over alphabet $\Lambda \cup S^y$. Since $\circledcirc$ composition is performed by applying $\circ$ to all t-assignments in $\widehat{X}$ the same holds for $\widehat{X} \circledcirc \widehat{Y}$.

We have derived this property for arbitrary $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S) \uparrow S$, which satisfy the preconditions mentioned above, so the same also holds for the both sides of Equation 3.26. If we mark the alphabet of variables in the rightmost composed transfer set as S^r then the result of the composition will have only expressions from language $(\Lambda \cup S^r)^+$, which equals $(\Lambda \cup S^z)^+$ in case of Equation 3.26.

We denote by $v \overset{s_i}{\rightsquigarrow} w$ replacing a symbol s_i in a string w by a string v. If all variable symbols are always replaced during concurrent substitution and boolean expressions are not optimized (i.e., a substitution algorithm works exactly according to Definition 3.17) then the associativity of transfer sets is similar to validity of the following equation:

$$\begin{aligned}
& \underline{(}.\lambda_{k_1}.s_k^z.\lambda_{k_2}.\underline{)} \overset{s_j^y}{\rightsquigarrow} \left(\underline{(}.\lambda_{j_1}.s_j^y.\lambda_{j_2}\underline{)} \overset{s_i^x}{\rightsquigarrow} \lambda_{i_1}.s_i^x.\lambda_{i_2} \right) \\
= \; & \left(\underline{(}.\lambda_{k_1}.s_k^z.\lambda_{k_2}.\underline{)} \overset{s_j^y}{\rightsquigarrow} \underline{(}.\lambda_{j_1}.s_j^y.\lambda_{j_2}\underline{)} \right) \overset{s_i^x}{\rightsquigarrow} \lambda_{i_1}.s_i^x.\lambda_{i_2}
\end{aligned} \tag{3.32}$$

Expanding the equation yields

$$\begin{aligned}
& \lambda_{i_1}.\underline{(}.\lambda_{j_1}.\underline{(}.\lambda_{k_1}.s_k^z.\lambda_{k_2}.\underline{)}.\lambda_{j_2}\underline{)}.\lambda_{i_2} \\
= \; & \lambda_{i_1}.\underline{(}.\lambda_{j_1}.\underline{(}.\lambda_{k_1}.s_k^z.\lambda_{k_2}.\underline{)}.\lambda_{j_2}\underline{)}.\lambda_{i_2}
\end{aligned} \tag{3.33}$$

Equation 3.32 could form the base case of induction with induction steps continuing by inserting new strings such that all expression strings will satisfy following conditions:

1. Single symbols of variables are separated by Λ^+ strings.
2. Exactly one substitution rule exists for each variable symbol.

This induction is trivial and therefore we allow to skip it.

Up to now, we have shown that associativity will hold if no logical manipulation with boolean expressions are performed after substitutions i.e., the substitution algorithm operates exactly according to Definition 3.17.

Possible modifications of the boolean expressions can absorb or add some variables but the result must be always logical equivalent to an original expression. Replacing each variable by an expression inclosed in parentheses preserves the precedence of all boolean operations and so such substitution is equivalent to replacing some variable by logical function. In this case. Boolean algebra assures the validity of its laws i.e.,

$$\begin{aligned}
tf &\equiv g(x_1, x_2, \dots x_n) \quad \text{implies} \\
tf &\equiv g(h_1(x_1, x_2, \dots x_n), h_2(x_1, x_2, \dots x_n), \dots h_n(x_1, x_2, \dots x_n))
\end{aligned}$$

where $g, h_1, h_2, \dots h_n$ are arbitrary logical functions of Boolean algebra and tf constant is either 1 or 0.

If an expression in a transfer set has been changed during composition then the same modification can also be done after replacing its variables by corresponding logical functions because whole substituted expression is enclosed in parentheses. The composition can yield different results in terms of languages but all belonging to the same class of the equivalence as show in the discussion of Equation 3.6 on page 55 and Equation 3.18 on page 63.

Therefore, Equation 3.26 holds even if some logical equivalent manipulations with transfer set expressions are performed.

□

Example 3.10

We return to three transfer sets utilized for proving that weak composition is not associative on $\widehat{\mathcal{B}}(S)$ (see Lemma 3.6 on page 58). Now, we test the same example, but with $\circledcirc$ composition.

$$\{\hat{x}(x)\,[\![x \wedge y]\!]\} \circledcirc \{\hat{y}(x)\,[\![x]\!]\ \} \circledcirc \{\hat{x}(x)\,[\![\neg x]\!]\}$$

We simplify the evaluation by applying Lemma 3.8 and Definition 3.24:

$$(\ \{\hat{x}(x)\,[\![x \wedge y]\!]\}\!\uparrow S \circledcirc \{\hat{y}(x)\,[\![x]\!]\ \}\!\uparrow S \circ \{\hat{x}(x)\,[\![\neg x]\!]\}\!\uparrow S)\!\downarrow$$

Expanding the equation according to Definition 3.20 yields

$$\left(\left\{\begin{array}{c}\hat{x}(x)\,[\![x \wedge y]\!]\,,\\ \hat{y}(y)\,[\![y]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{x}(x)\,[\![x]\!]\,,\\ \hat{y}(x)\,[\![x]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{x}(x)\,[\![\neg x]\!]\,,\\ \hat{y}(y)\,[\![y]\!]\end{array}\right\}\right)\!\downarrow$$

We will test associativity first by composing two leftmost terms which yields

$$\begin{array}{rl} & \left(\left\{\begin{array}{c}\hat{x}(x)\,[\![x \wedge x]\!]\,,\\ \hat{y}(x)\,[\![x]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{x}(x)\,[\![\neg x]\!]\,,\\ \hat{y}(y)\,[\![y]\!]\end{array}\right\}\right)\!\downarrow \\ = & \left\{\begin{array}{c}\hat{x}(x)\,[\![\neg x]\!]\,,\\ \hat{y}(x)\,[\![\neg x]\!]\end{array}\right\}\!\downarrow \ = \left\{\begin{array}{c}\hat{x}(x)\,[\![\neg x]\!]\,,\\ \hat{y}(x)\,[\![\neg x]\!]\end{array}\right\} = \widehat{\Theta}(\neg x)^S \end{array}$$

where $\widehat{\Theta}()^S$ was define by Equation 3.19 on page 63. Now we begin with two rightmost terms which give the equations:

$$\begin{array}{rl} & \left(\left\{\begin{array}{c}\hat{x}(x)\,[\![x \wedge y]\!]\,,\\ \hat{y}(y)\,[\![y]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{x}(x)\,[\![\neg x]\!]\,,\\ \hat{y}(x)\,[\![\neg x]\!]\end{array}\right\}\right)\!\downarrow \\ = & \left\{\begin{array}{c}\hat{x}(x)\,[\![\neg x \wedge \neg x]\!]\,,\\ \hat{y}(x)\,[\![\neg x]\!]\end{array}\right\}\!\downarrow \ = \left\{\begin{array}{c}\hat{x}(x)\,[\![\neg x]\!]\,,\\ \hat{y}(x)\,[\![\neg x]\!]\end{array}\right\} = \widehat{\Theta}(\neg x)^S \end{array}$$

The example which has failed to be associative for $\circ$ weak composition is associative when evaluated as $\circledcirc$ composition, since all variables propagate to the final result.

Proposition 3.9 $\widehat{\mathbb{M}}(S) = \left(\widehat{\mathcal{S}}(S)/\widehat{=}, \widehat{=}\,, \circledcirc\,, \widehat{\mathcal{E}}^S\right)$ *is a monoid.*

Proof: We have proved that $\circledcirc$ is an associative operation on $\widehat{\mathcal{S}}(S)$ in the previous proposition. To prove that $\widehat{\mathcal{S}}(S)$ is closed under $\circledcirc$, we utilize that the semigroup

$$\widehat{\mathbb{G}}(S/x) = (\widehat{\mathcal{S}}(S/x)/\widehat{=}\,, \circ\,)$$

is closed under $\circ$ composition as shown in Proposition 3.4 (see page 59).

Let $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ be transfer sets then their composition $\widehat{X} \circledcirc \widehat{Y}$ is performed as a consecutive $\circ$ compositions of t-assignments from $\widehat{X}$ and $\widehat{Y}$, which belong to an identical variable (see Definition 3.24 and also Equation 3.27), because the fact that $\widehat{\mathbb{G}}(S/x)$ semigroup is closed under $\circ$ implies that $\widehat{\mathcal{S}}(S)$ is also closed under $\circledcirc$.

Now, we show that $\widehat{\mathcal{E}}^S$ is an identity element on $\widehat{\mathcal{S}}(S)$.

$$\widehat{X} \circledcirc \widehat{\mathcal{E}}^S \mathrel{\widehat{=}} \widehat{\mathcal{E}}^S \circledcirc \widehat{X} \tag{3.34}$$

We prove the equation only for all $\widehat{X} \in \widehat{\mathcal{S}}(S) \uparrow S$ because the same will hold for any transfer set according Lemma 3.8. Using definition of $\widehat{\mathcal{E}}^S$ (see page 61) we expand Equation 3.34 to

$$\left\{\begin{array}{l} \hat{x}_1 \llbracket bexp_1 \rrbracket, \\ \hat{x}_2 \llbracket bexp_2 \rrbracket, \\ \dots \\ \hat{x}_n \llbracket bexp_2 \rrbracket \end{array}\right\} \circledcirc \left\{\begin{array}{l} \hat{x}_1 \llbracket x_1 \rrbracket, \\ \hat{x}_2 \llbracket x_2 \rrbracket, \\ \dots \\ \hat{x}_n \llbracket x_n \rrbracket \end{array}\right\} \mathrel{\widehat{=}} \left\{\begin{array}{l} \hat{x}_1 \llbracket x_1 \rrbracket, \\ \hat{x}_2 \llbracket x_2 \rrbracket, \\ \dots \\ \hat{x}_n \llbracket x_n \rrbracket \end{array}\right\} \circledcirc \left\{\begin{array}{l} \hat{x}_1 \llbracket bexp_1 \rrbracket, \\ \hat{x}_2 \llbracket bexp_2 \rrbracket, \\ \dots \\ \hat{x}_n \llbracket bexp_2 \rrbracket \end{array}\right\}$$

It yields after applying Lemma 3.8 to the leftmost side and Definition 3.24 to rightmost side

$$\left\{\begin{array}{l} \hat{x}_1 \llbracket bexp_1 \rrbracket, \\ \hat{x}_2 \llbracket bexp_2 \rrbracket, \\ \dots \\ \hat{x}_n \llbracket bexp_2 \rrbracket \end{array}\right\} \circledcirc \emptyset \mathrel{\widehat{=}} \left\{\begin{array}{l} \hat{x}_1 \llbracket bexp_1 \rrbracket, \\ \hat{x}_2 \llbracket bexp_2 \rrbracket, \\ \dots \\ \hat{x}_n \llbracket bexp_2 \rrbracket \end{array}\right\}$$

that certainly holds because empty set means that nothing is substituted. Therefore the both side are $\widehat{=}$ equal [12] of expressions and $\widehat{\mathcal{E}}^S$ is identity element on $\widehat{\mathcal{S}}(S)$.

□

Associativity with the existence of identical element allow presenting one definition and several propositions, which will become useful for creating the conversion of APLC program in the next section.

Definition 3.25 *Let $\widehat{X}_i \in \widehat{\mathcal{S}}(S)$ be n transfer sets on S where $n > 0$ is an integer. To shorten notations, let us write sequences of their compositions with the aid of* compositional production *defined as the follows:*

$$\widehat{\prod}_{i=1}^{n} \widehat{X}_i \stackrel{df}{=} \widehat{X}_k \circledcirc \widehat{X}_{k+1} \circledcirc \dots \widehat{X}_n$$
$$\widehat{\prod}_{i=n}^{k<} \widehat{X}_i \stackrel{df}{=} \widehat{X}_n \circledcirc \widehat{X}_{n-1} \circledcirc \dots \widehat{X}_k$$

where $k > 0$ is an integer number such that $k < n$.

[12] The concurrent substitution algorithm used in the compositions allows any result equivalent to Definition 3.17 on page 56, therefore Equation 3.34 only specifies $\widehat{=}$ equality.

Using reverse order of indexes in the second compositional production reflexes composing transfer sets in the direction from right to left.

Proposition 3.10 *Let $\widehat{X}_i \in \widehat{\mathcal{S}}(S)$ be n transfer sets on S, where $n > 0$ is an integer. Their composition can be evaluated as the composition of two partial compositions*

$$\widehat{\prod}_{i=n}^{1<} \widehat{X}_i \mathrel{\widehat{=}} \left(\widehat{\prod}_{i=n}^{k+1<} \widehat{X}_i \right) \circledcirc \left(\widehat{\prod}_{i=k}^{1<} \widehat{X}_i \right) \tag{3.35}$$

for any k integer number such that $1 < k < n$.

Proof: The proposition follows directly from associativity of $\circledcirc$ operation proven above. $\square$

Proposition 3.11 *Let $\widehat{X}_i, \widehat{Y}_j \in \widehat{\mathcal{S}}(S)$ be m and n transfer sets on S, where $m > 0$ and $n > 0$ are integers, and given their composition*

$$\widehat{X} = \widehat{\prod}_{i=m}^{1<} \widehat{X}_i \quad \textit{and} \quad \widehat{Y} = \widehat{\prod}_{j=n}^{1<} \widehat{Y}_j$$

and the formula for the evaluating transfer set $\widehat{Z} \in \widehat{\mathcal{S}}(S)$ as the following branching condition:

$$\widehat{Z} = \left\{ \begin{array}{ll} \widehat{X} & \textit{if } b \equiv 1 \\ \widehat{Y} & \textit{if } b \equiv 0 \end{array} \right.$$

where $b \in S$ is a variable. Then, $\widehat{Z}$ equals to

$$\widehat{Z} = \left(\widehat{X} \wedge \widehat{\Theta}(b)^S \right) \vee \left(\widehat{Y} \wedge \widehat{\Theta}(\neg b)^S \right) \tag{3.36}$$

Proof: Expanding Equation 3.36 according to Definition 3.23 and Equation 3.19 (see page 63) we obtain (notice we defined here the precedence of wedged operator $\wedge$ higher than $\vee$ to wedge the equation at the line):

$$\widehat{Z} = \left\{ \begin{array}{c} \hat{s}_1^x \, [\![bexp_1^x]\!] \\ \hat{s}_2^x \, [\![bexp_2^x]\!] \\ \dots \\ \hat{s}_n^x \, [\![bexp_n^x]\!] \end{array} \right\} \wedge \left\{ \begin{array}{c} \hat{s}_1^b \, [\![b]\!] \\ \hat{s}_2^b \, [\![b]\!] \\ \dots \\ \hat{s}_n^b \, [\![b]\!] \end{array} \right\} \vee \left\{ \begin{array}{c} \hat{s}_1^y \, [\![bexp_1^y]\!] \\ \hat{s}_2^y \, [\![bexp_2^y]\!] \\ \dots \\ \hat{s}_n^y \, [\![bexp_n^y]\!] \end{array} \right\} \wedge \left\{ \begin{array}{c} \hat{s}_1^{\bar{b}} \, [\![\neg b]\!] \\ \hat{s}_2^{\bar{b}} \, [\![\neg b]\!] \\ \dots \\ \hat{s}_n^{\bar{b}} \, [\![\neg b]\!] \end{array} \right\}$$

where $n = |S|$. This yields

$$\widehat{Z} = \left\{ \begin{array}{c} \hat{s}_1 \, [\![((bexp_1^x) \wedge b) \vee ((bexp_1^y) \wedge \neg b)]\!] \\ \hat{s}_2 \, [\![((bexp_2^x) \wedge b) \vee ((bexp_2^y) \wedge \neg b)]\!] \\ \dots \\ \hat{s}_n \, [\![((bexp_n^x) \wedge b) \vee ((bexp_n^y) \wedge \neg b)]\!] \end{array} \right\}$$

from which we see that

$$\widehat{X} = \widehat{Z} \circledcirc \left\{ \hat{b}[\![1]\!] \right\} \quad \text{and} \quad \widehat{Y} = \widehat{Z} \circledcirc \left\{ \hat{b}[\![0]\!] \right\} \tag{3.37}$$

□

The last proposition combines the previous conclusions and will be useful for converting APLC programs.

Proposition 3.12 *Let $\widehat{X}_i, \widehat{Y}_j, \widehat{W}_k, \in \widehat{\mathcal{S}}(S)$ be transfer sets on S and integer numbers satisfying $0 < i \le m$, $0 < j \le n$, $0 < k \le r$, and $0 < l \le s$. Given their compositions*

$$\widehat{V} = \widehat{\prod}_{k=r}^{1<} \widehat{V}_k \quad \textit{and} \quad \widehat{W} = \widehat{\prod}_{l=s}^{1<} \widehat{W}_l \tag{3.38}$$

$$\widehat{X} = \widehat{V} \circledcirc \left(\widehat{\prod}_{i=m}^{1<} \widehat{X}_i \right) \circledcirc \widehat{W} \tag{3.39}$$

$$\widehat{Y} = \widehat{V} \circledcirc \left(\widehat{\prod}_{j=n}^{1<} \widehat{Y}_j \right) \circledcirc \widehat{W} \tag{3.40}$$

and the formula for evaluating transfer set $\widehat{Z} \in \widehat{\mathcal{S}}(S)$ as the following branching condition:

$$\widehat{Z} = \begin{cases} \widehat{X} & \textit{if } b \equiv 1 \\ \widehat{Y} & \textit{if } b \equiv 0 \end{cases} \tag{3.41}$$

where $b \in S$ is a variable. Then, $\widehat{Z}$ equals

$$\widehat{Z} = \widehat{V} \circledcirc \left(\left(\widehat{X} \wedge \widehat{\Theta}(b)^S \right) \vee \left(\widehat{Y} \wedge \widehat{\Theta}(\neg b)^S \right) \right) \circledcirc \widehat{W} \tag{3.42}$$

Proof: Substituting Equations 3.37 into Equations 3.39 and 3.40 we obtain

$$\begin{aligned} \widehat{X} = &\ \widehat{V} \circledcirc \left(\left(\left(\widehat{X} \wedge \widehat{\Theta}(b)^S \right) \vee \left(\widehat{Y} \wedge \widehat{\Theta}(\neg b)^S \right) \right) \circledcirc \left\{ \hat{b}[\![1]\!] \right\} \right) \circledcirc \widehat{W} \\ \widehat{Y} = &\ \widehat{V} \circledcirc \left(\left(\left(\widehat{X} \wedge \widehat{\Theta}(b)^S \right) \vee \left(\widehat{Y} \wedge \widehat{\Theta}(\neg b)^S \right) \right) \circledcirc \left\{ \hat{b}[\![0]\!] \right\} \right) \circledcirc \widehat{W} \end{aligned}$$

Applying the condition presented in Equation 3.41 yields

$$\begin{aligned} \widehat{Z} &= \widehat{V} \circledcirc \left(\left(\left(\widehat{X} \wedge \widehat{\Theta}(b)^S \right) \vee \left(\widehat{Y} \wedge \widehat{\Theta}(\neg b)^S \right) \right) \circledcirc \left\{ \hat{b}[\![b]\!] \right\} \right) \circledcirc \widehat{W} \\ &= \widehat{V} \circledcirc \left(\left(\left(\widehat{X} \wedge \widehat{\Theta}(b)^S \right) \vee \left(\widehat{Y} \wedge \widehat{\Theta}(\neg b)^S \right) \right) \circledcirc \widehat{\mathcal{E}}^S \right) \circledcirc \widehat{W} \\ &= \widehat{V} \circledcirc \left(\left(\widehat{X} \wedge \widehat{\Theta}(b)^S \right) \vee \left(\widehat{Y} \wedge \widehat{\Theta}(\neg b)^S \right) \right) \circledcirc \widehat{W} \end{aligned}$$

which is Equation 3.42. □

3.4 Converting APLC Program

Here we explain APLCTRANS, the an effective algorithm for representing APLC program operations as one transfer set.

APLCTRANS supposes expressing all instructions of APLC program as transfer sets whose definition requires a storage S. S incorporates for binary PLC (see Definition 3.2 on page 32) three sets of variables:

Σ - finite set of inputs,

Ω - finite set of outputs, and

V - internal memory of binary PLC,

to which we must also add the boolean flag register $\hat{f}_{reg}$ and an evaluation stack E_{stack} utilized by APLC machine. We will suppose a finite length of E_{stack} specified by an integer constant *depth* (see Table 3.7). If a chosen *depth* appears to be insufficient, its increasing is possible.

Transfer sets will be defined on the union of all mentioned variables:

$$S = \Sigma \cup \Omega \cup V \cup \{\hat{f}_{reg}\} \cup E_{stack} \tag{3.43}$$

Converting a real PLC program begins by expressing its instructions in APLC language. This easily managed, since both APLC language and machine were designed to allow such conversions as effective and readily as possible.

The outline of some simple PLC instructions with direct conversions is listed in Table 3.6. More complex program constructions require certainly special approaches depending on an employed PLC processor, but in general the most of PLC instructions are convertible by substitutions of some strings. This is the main goal of APLC machine described in Section 3.2.

The transfer sets of APLC instructions follow directly from joining Tables 3.3 and 3.1 (see pages 42 and 38). They are listed in Table 3.7.

3.4.1 APLCTRANS - Algorithm

Preparation of data
When APLCTRANS algorithm runs, APLC instructions are replaced by corresponding transfer sets. We will index them in the physical order of the instructions in the program. If an instruction was labeled, then its transfer set is marked by the same label.

The first instruction, from which the program begins, may be labeled, but not necessarily, because jumps and call are not allowed to address any label before them due to the constraint specified by $\prec$ relation (see page 42 and Table 3.3).

Allen-Bradley PLC 5 and SLC 500		
Rung start and end		
SOR	$\rightarrow$	INIT;
EOR	$\rightarrow$	INIT;
Branch start, next, or end		
BST	$\rightarrow$	PUSH; EPUSH 1;
BNX	$\rightarrow$	TOR; DUP; EXCH;
BND	$\rightarrow$	TOR; DROP;
Examine if closed or opened		
XIC b	$\rightarrow$	AND b;
XIO b	$\rightarrow$	AND $\neg b$
Output energize, latch, or unlatch		
OTE b	$\rightarrow$	STORE b;
OTL b	$\rightarrow$	SET b
OTU b	$\rightarrow$	RES b
Siemens S7-200		
Rung start and end		
Network	$\rightarrow$	INIT;
Logic stack push, read, pop, load		
LPS	$\rightarrow$	PUSH;
LRD	$\rightarrow$	LOAD 1; TAND;
LPP	$\rightarrow$	POP;
LDS b	$\rightarrow$	EPUSH b;
And, And not, Or, Or not, Load, and Load-not		
A b	$\rightarrow$	AND b;
AN b	$\rightarrow$	AND $\neg b$
O b	$\rightarrow$	OR b;
ON b	$\rightarrow$	OR $\neg b$
L b	$\rightarrow$	LOAD b;
LN b	$\rightarrow$	LOAD $\neg b$
Assign, set, reset		
= b	$\rightarrow$	STORE b;
S b_1, n	$\rightarrow$	SET b_1; SET b_2; ... SET b_n;
R b_1, n	$\rightarrow$	RES b_1; RES b_2; ... RES b_n;

Table 3.6: Examples of Converting PLC Codes into APLC

Initialization of evaluation			
INIT	$\widehat{C}_{\text{Init}}$	=	$\{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$
Operation with flag register			
LOAD *bexp*	$\widehat{C}_{\text{Load}}$	=	$\{\ \hat{f}_{reg}\,[\![bexp]\!]\ \}$
AND *bexp*	$\widehat{C}_{\text{And}}$	=	$\{\ \hat{f}_{reg}\,[\![f_{reg} \wedge (bexp)]\!]\ \}$
OR *bexp*	$\widehat{C}_{\text{Or}}$	=	$\{\ \hat{f}_{reg}\,[\![f_{reg} \vee (bexp)]\!]\ \}$
NOT	$\widehat{C}_{\text{Not}}$	=	$\{\ \hat{f}_{reg}\,[\![\neg f_{reg}]\!]\ \}$
TAND	$\widehat{C}_{\text{TAnd}}$	=	$\{\ \hat{f}_{reg}\,[\![f_{reg} \wedge e_1]\!]\ \}$
TOR	$\widehat{C}_{\text{TOr}}$	=	$\{\ \hat{f}_{reg}\,[\![f_{reg} \vee e_1]\!]\ \}$
Assignments			
STORE *b*	$\widehat{C}_{\text{Store}}$	=	$\{\ \hat{b}\,[\![f_{reg}]\!]\ \}$
SET *b*	$\widehat{C}_{\text{Set}}$	=	$\{\ \hat{b}\,[\![f_{reg} \vee b]\!]\ \}$
RES *b*	$\widehat{C}_{\text{Res}}$	=	$\{\ \hat{b}\,[\![\neg f_{reg} \wedge b]\!]\ \}$
Rising and falling edge detection			
REDGE *b*	$\widehat{C}_{\text{REdge}}$	=	$\{\ \hat{f}_{reg}\,[\![f_{reg} \wedge \neg b]\!]\,,\hat{b}\,[\![f_{reg}]\!]\ \}$
FEDGE *b*	$\widehat{C}_{\text{FEdge}}$	=	$\{\ \hat{f}_{reg}\,[\![\neg f_{reg} \wedge b]\!]\,,\hat{b}\,[\![f_{reg}]\!]\ \}$
Program control (see text)			
JP *label*	$\widehat{C}_{\text{Jp}}$	=	$\widehat{C}_{label}$
JPC *label*	$\widehat{C}_{\text{Jpc}}$	=	Equation 3.42 or 3.36
JS *label*	$\widehat{C}_{\text{Js}}$	=	$\widehat{C}_{nextcode} \circledcirc \widehat{C}_{label}$
JSC *label*	$\widehat{C}_{\text{Jsc}}$	=	Equation 3.42
END	$\widehat{C}_{\text{End}}$	=	$\{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$ *and the end of current composition*

Table continues in Table 3.8

Table 3.7: Transfer Sets of APLC Program — Part I.

Table continues from Table 3.7			
Manipulation with the evaluation stack			
PUSH	$\widehat{C}_{\text{Push}}$	$=$	$\{\ \hat{e}_1\,[\![f_{reg}]\!]\ \} \cup \widehat{E}_{push}$
EPUSH *bexp*	$\widehat{C}_{\text{EPush}}$	$=$	$\{\ \hat{e}_1\,[\![bexp]\!]\ \} \cup \widehat{E}_{push}$
POP	$\widehat{C}_{\text{Pop}}$	$=$	$\{\ \hat{f}_{reg}\,[\![e_1]\!]\ \} \cup \widehat{E}_{pop}$
DROP	$\widehat{C}_{\text{Drop}}$	$=$	$\widehat{E}_{pop}$
DUP	$\widehat{C}_{\text{Dup}}$	$=$	$\{\ \hat{e}_2\,[\![e_1]\!]\ \} \cup \left(\widehat{E}_{push} - \hat{e}_2\right)$
FSWP	$\widehat{C}_{\text{FSwp}}$	$=$	$\{\ \hat{f}_{reg}\,[\![e_1]\!]\,, \hat{e}_1\,[\![f_{reg}]\!]\ \}$
ESWP	$\widehat{C}_{\text{ESwp}}$	$=$	$\{\ \hat{e}_1\,[\![e_2]\!]\,, \hat{e}_2\,[\![e_1]\!]\ \}$
Special transfer sets for the evaluation stack			
E_{stack}	$=$	$\{e_1, e_2, e_3,\ \dots\ , e_{depth}\}$	
$\widehat{E}_{push}$	$=$	$\{\hat{e}_{i+1}\,[\![e_i]\!]\ \mid e_i \in E_{stack} - \{e_{depth}\}\}$	
$\widehat{E}_{pop}$	$=$	$\{\hat{e}_i\,[\![e_{i+1}]\!]\ \mid e_i \in E_{stack} - \{e_{depth}\}\} \cup \{\hat{e}_{depth}\,[\![0]\!]\}$	
		where $depth > 1$ is an integer and $\text{co}(\hat{e}_i) = e_i$	

Table 3.8: Transfer Sets of APLC Program — Part II.

If the first instruction is not labeled, then we mark it by some unique identifier, for example '*Label1*', to obtain a program containing $k \geq 1$ instructions marked by labels.

We divide the instructions into k block with lengths $m_1, m_2, \dots\ m_k$. Block i begins by $label_i$ as shown in Table 3.9 (see page 77) and contains at least END instruction, so $m_i \geq 1$ for all $i \in |I|$, $|I| = k$.

We index the transfer sets of individual instructions as all being listed in k lists

$$\begin{aligned}
L_1 &= \widehat{C}_{1,1} :: \widehat{C}_{1,2} :: \dots :: \widehat{C}_{1,m_1} \\
L_2 &= \widehat{C}_{2,1} :: \widehat{C}_{2,2} :: \dots :: \widehat{C}_{2,m_2} \\
& \quad \dots \\
L_{m_{k-1}} &= \widehat{C}_{k-1,1} :: \widehat{C}_{k-1,2} :: \dots :: \widehat{C}_{k-1,m_{k-1}} \\
L_{m_k} &= \widehat{C}_{m_k,1} :: \widehat{C}_{m_k,2} :: \dots :: \widehat{C}_{m_k,m_k}
\end{aligned} \tag{3.44}$$

where $\widehat{C}_{1,1}$ represent the transfer set of the first instruction of the program, $\widehat{C}_{m_k,m_k}$ is the last one (in terms of their physical placement in program's source). We obtain at least one list L_1.

The conversion of APLC instructions can be surely performed at runtime, because it is a simple replacement of text strings by corresponding

$$\begin{array}{lll}
\mathit{label}_1 & : & \left.\begin{array}{l}\widehat{C}_{1,1}\\ \widehat{C}_{1,2}\\ \dots\\ \widehat{C}_{1,m_1}\end{array}\right\}\widehat{C}_1\\[2ex]
\mathit{label}_2 & : & \left.\begin{array}{l}\widehat{C}_{2,1}\\ \widehat{C}_{2,2}\\ \dots\\ \widehat{C}_{2,m_2}\end{array}\right\}\widehat{C}_2\\[2ex]
 & & \dots\\[2ex]
\mathit{label}_{k-1} & : & \left.\begin{array}{l}\widehat{C}_{k-1,1}\\ \widehat{C}_{k-1,2}\\ \dots\\ \widehat{C}_{k-1,m_{k-1}}\end{array}\right\}\widehat{C}_{k-1}\\[2ex]
\mathit{label}_k & : & \left.\begin{array}{l}\widehat{C}_{k,1}\\ \widehat{C}_{k,2}\\ \dots\\ \widehat{C}_{k,m_k}\end{array}\right\}\widehat{C}_k = \widehat{\prod}_{i=m_k}^{1<}\widehat{C}_{k,i}
\end{array}$$

Table 3.9: Diagram of APLC Program Composition

transfer sets definitions taken from a short table. The lists presented above only clarify used indexing in the text below.

Composing the last block
After preparing data, APLCTRANS begins by composing k-th block (the last one) which does not contain any control instruction due to constraints mentioned above, so $\widehat{C}_k$ equals

$$\widehat{C}_k := \widehat{\prod}_{i=m_k}^{1<}\widehat{C}_{k,i} \tag{3.45}$$

The other blocks are composed in the order from $k-1$ to 1 in the following loop.

LOOP: for $\mathit{ilist} := k-1$ **downto** 1 **do**
The transfer set of *ilist* block is calculated as

$$\widehat{C}_{\mathit{ilist}} := \textbf{ComposeTS}(\mathit{ilist}, 1);$$

where ComposeTS($\mathit{ilist}, 1$) is recursive function whose arguments are described in Pascal like syntax below. LOOP is repeated until $\mathit{ilist} = 1$.

End of LOOP
Transfer set $\widehat{C}_1$ expresses the operations of whole APLC program.

---

function ComposeTS($\mathit{ilist}, \mathit{jset}$: integer) : TransferSet**;**

First, the internal variable of **ComposeTS** is initialized $\widehat{C} := \widehat{\mathcal{E}}^S$. It will store partial results during the computation of transfer set of *ilist* block from $\widehat{C}_{\mathit{ilist},\mathit{jset}}$ to $\widehat{C}_{\mathit{ilist},m_{\mathit{ilist}}}$ in the following for-loop:

LOOPFN: for $j := \mathit{jset}$ **to** m_{ilist} **do**

Inside of LOOPFN program block, we switch according to the type of current instruction that was converted to transfer set $\widehat{C}_{\mathit{ilist},j}$.

case End: LOOPFN is interrupted and the function returns a value of $\widehat{C} := \Big(\{\ \hat{f}_{\mathit{reg}}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E\Big) \circledcirc \widehat{C}$.

case Js: $\widehat{C}$ is updated to $\widehat{C} := \widehat{C}_{\mathit{js-label}} \circledcirc \widehat{C}$ where $\widehat{C}_{\mathit{js-label}}$ represents a block designated by Js operand. This block has already been evaluated due to the constraint for jumps and calls mentioned above. Then, LOOPFN continues.

case Jsc: Let us write f'_{reg} for momentary value of $\hat{f}_{\mathit{reg}} \mathrel{\widehat{\in}} \widehat{C}$ when Jsc was encountered. When Jsc call is not invoked then $\widehat{\mathcal{E}}^S$ represents its operations. Substituting into Equation 3.42 defines updating $\widehat{C}$:

$$\widehat{C} \;:=\; \Big(\Big(\widehat{C}_{\mathit{js-label}} \wedge \widehat{\Theta}(f'_{\mathit{reg}})^S\Big) \vee \Big(\widehat{\mathcal{E}}^S \wedge \widehat{\Theta}(\neg f'_{\mathit{reg}})^S\Big)\Big) \circledcirc \widehat{C}$$

After updating $\widehat{C}$, LOOPFN continues.

case Jp: LOOPFN is interrupted and the function returns

$$\widehat{C} := \widehat{C}_{\mathit{jp-label}} \circledcirc \widehat{C}$$

where $\widehat{C}_{\mathit{jp-label}}$ represents already evaluated block designated by Jp label. LOOPFN does not continue because Jp is either followed immediately by $\mathit{ilist} + 1$ block or by some never executed instructions.

case Jpc: Let us write f'_{reg} for momentary value of $\hat{f}_{\mathit{reg}} \mathrel{\widehat{\in}} \widehat{C}$. Then, LOOPFN is interrupted and the function returns the value of Equation 3.42

$$\widehat{C} := \Big(\Big(\widehat{C}_{\mathit{jp-label}} \wedge \widehat{\Theta}(f'_{\mathit{reg}})^S\Big) \vee \Big(\widehat{C}_{\mathit{jp-next}} \wedge \widehat{\Theta}(\neg f'_{\mathit{reg}})^S\Big)\Big) \circledcirc \widehat{C}$$

where $\widehat{C}_{jp-next}$ is the transfer set of next operations i.e., after this jump instruction, that are given by the recursive call:

$$\widehat{C}_{jp-next} := \textbf{ComposeTS}(\mathit{ilist}, j+1)$$

Else: $\widehat{C}$ is updated to $\widehat{C} := \widehat{C}_{\mathit{ilist},\mathit{jset}} \circledcirc \widehat{C}$ and then LOOPFN continues.

End of LOOPFN If the loop has not been terminated prematurely and reached this point, then block *ilist* has no End instruction and continues to block $\mathit{ilist}+1$. Therefore, the function returns a value of the composition $\widehat{C} := \widehat{C}_{\mathit{ilist}+1} \circledcirc \widehat{C}$.

Remark to algorithm: The APLCTRANS algorithm described above composes the instructions in two directions. It takes program blocks in backward order, but it proceeds the instructions of momentary block in forward direction.

The composition can be performed only in the backward direction, from a bottom instruction to a top one, because $\circledcirc$ composition is associative. In this case, the recursive call will not be required.

But we have intentionally described the algorithm in the form, in which it was really tested. The strict backward direction appeared too unclear for debugging. To simplify experiments, APLCTRANS reverses only in blocks and moves forward when composing instructions of one block.

Example 3.11

Figure 3.2 displays the program written under developing environment RSLogix 5 for PLC-5 family of PLCs produced by Rockwell Automation.

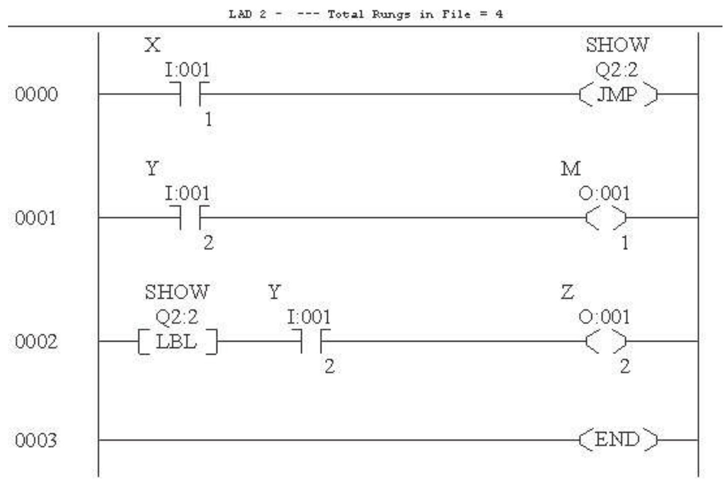
The middle part of the figure gives the listing of PLC processor codes. RSLogix 5 allows editing these instructions in the graphical form of the ladder diagram depicted at the top of the figure. The corresponding transfer sets and APLC instructions are listed at the bottom. We will suppose that $\Sigma = \{x, y\}$, $\Omega = \{z\}$, and $V = \{m\}$. This definition is also necessary input information.

The example is analogous to Pascal program: [13]

```
var x, y, z, m: boolean;
begin   if not x then m:=y;
        y:=z;
end;
```

[13]The presented example is very bad PLC program that was created intentionally only for demonstrating Jpc composition — the most complex case. The example could be programmed by much better ways, but such programs have appeared too unreadable for manual composition.

Listing created by PLC development environment RSLogix 5



```
PROJECT "EXAMPLE"                    ADDRESS..SYMBOL
LADDER 2                             O:001/2    Z
% Rung: 0 %                          O:001/1    M
SOR XIC I:001/1 JMP 2 EOR            I:001/2    Y
% Rung: 1 %                          I:001/1    X
SOR XIC I:001/2 OTE O:001/1 EOR
% Rung: 2 %
SOR LBL 2 XIC I:001/2 OTE O:001/2 EOR
```

PLC	APLC	Transfer set
SOR	INIT	$\widehat{C}_{1,1} = \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$
XIC X	AND x	$\widehat{C}_{1,2} = \{\ \hat{f}_{reg}\,[\![f_{reg} \wedge x]\!]\ \}$
JMP 2	JPC $block_2$	$\widehat{C}_{1,3} =$ Equation 3.42
EOR	INIT	$\widehat{C}_{1,4} = \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$
SOR	INIT	$\widehat{C}_{1,5} = \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$
XIC Y	AND y	$\widehat{C}_{1,6} = \{\ \hat{f}_{reg}\,[\![f_{reg} \wedge y]\!]\ \}$
OTE M	STORE m	$\widehat{C}_{1,7} = \{\ \hat{m}\,[\![f_{reg}]\!]\ \}$
EOR	INIT	$\widehat{C}_{1,8} = \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$
SOR	INIT	$\widehat{C}_{1,9} = \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$
LBL 2	$block_2$:	
XIC Y	AND y	$\widehat{C}_{2,1} = \{\ \hat{f}_{reg}\,[\![f_{reg} \wedge y]\!]\ \}$
OTE Z	STORE z	$\widehat{C}_{2,2} = \{\ \hat{z}\,[\![f_{reg}]\!]\ \}$
EOR	INIT	$\widehat{C}_{2,3} = \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$
end of file	END	$\widehat{C}_{2,4} = \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E$

Figure 3.2: Example of Composing PLC Program

The program does not utilize evaluation stack, thus its storage is according to Equation 3.43 (see page 73) $S = \Sigma \cup V \cup \Omega = \{x, y, z, m, f_{reg}\}$.

Omitting unused manipulation with the evaluation stack gives the composition of the last block as the following:

$$\begin{aligned}
\widehat{C}_2 &= \widehat{C}_{2,3} \circledcirc \widehat{C}_{2,3} \circledcirc \widehat{C}_{2,2} \circledcirc \widehat{C}_{2,1} \\
&= \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \circledcirc \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \circledcirc \{\ \hat{z}\,[\![f_{reg}]\!]\ \} \circledcirc \{\ \hat{f}_{reg}\,[\![f_{reg} \wedge y]\!]\ \} \\
&= \{\ \hat{f}_{reg}\,[\![1]\!], \hat{z}\,[\![f_{reg} \wedge y]\!]\ \}
\end{aligned}$$

After the last block, the previous one is computed by invoking $\mathbf{ComposeTS}(1,1)$, which composes two instructions of bock 1:

$$\begin{aligned}
\widehat{C}_{1a} &= \widehat{C}_{1,2} \circledcirc \widehat{C}_{1,1} \\
&= \{\ \hat{f}_{reg}\,[\![f_{reg} \wedge x]\!]\ \} \circledcirc \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \\
&= \{\ \hat{f}_{reg}\,[\![x]\!]\ \}
\end{aligned}$$

but the third instruction is JSC branching operation. To compose it, **ComposeTS** must first evaluate the transfer sets of codes after JSC as $\mathbf{ComposeTS}(1,4)$.

The recursive call evaluates:

$$\begin{aligned}
\widehat{C}_{1b} &= \widehat{\prod}_{i=9}^{4<} \widehat{C}_{1,i} \\
&= \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \circledcirc \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \circledcirc \{\ \hat{m}\,[\![f_{reg}]\!]\ \} \qquad (3.46) \\
&\quad \circledcirc \{\ \hat{f}_{reg}\,[\![f_{reg} \wedge y]\!]\ \} \circledcirc \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \circledcirc \{\ \hat{f}_{reg}\,[\![1]\!]\ \} \\
&= \{\ \hat{f}_{reg}\,[\![1]\!], \hat{m}\,[\![y]\!]\ \}
\end{aligned}$$

Because LOOPFN was not interrupted by END, block 1 continues to block 2 and we must compose them together:

$$\begin{aligned}
\widehat{C}_{next} &= \widehat{C}_2 \circledcirc \widehat{C}_{1b} \\
&= \{\ \hat{f}_{reg}\,[\![1]\!], \hat{z}\,[\![f_{reg} \wedge y]\!]\ \} \circledcirc \{\ \hat{f}_{reg}\,[\![1]\!], \hat{m}\,[\![f_{reg} \wedge y]\!]\ \} \\
&= \{\ \hat{f}_{reg}\,[\![1]\!], \hat{z}\,[\![y]\!], \hat{m}\,[\![y]\!]\ \}
\end{aligned}$$

The recursive call returns this value.

After obtaining this result, main **ComposeTS** evaluates JSC as:

$$\widehat{C}_1 = \widehat{C}_{jpc} \circledcirc \widehat{C}_{1a} = \left(\left(\widehat{C}_2 \wedge \widehat{\Theta}(x)^S\right) \vee \left(\widehat{C}_{next} \wedge \widehat{\Theta}(\neg x)^S\right)\right) \circledcirc \widehat{C}_{1a}$$

where x is the value of $\hat{f}_{reg} \mathrel{\widehat{\in}} \widehat{C}_{1a}$ i.e., the momentary value of the flag register when JPC was encountered. The substitution of $\widehat{C}_{next}$ and $\widehat{C}_2$ yields

$$\widehat{C}_{jpc} = \left(\left\{ \begin{array}{c} \hat{f}_{reg}\,[\![1]\!], \\ \hat{z}\,[\![f_{reg} \wedge y]\!] \end{array} \right\} \wedge \widehat{\Theta}(x)^S\right) \vee \left(\left\{ \begin{array}{c} \hat{f}_{reg}\,[\![1]\!], \\ \hat{z}\,[\![y]\!], \\ \hat{m}\,[\![y]\!] \end{array} \right\} \wedge \widehat{\Theta}(\neg x)^S\right)$$

$$= \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket x \rrbracket, \\ \hat{x} \llbracket x \rrbracket, \\ \hat{y} \llbracket y \wedge x \rrbracket, \\ \hat{z} \llbracket f_{reg} \wedge y \wedge x \rrbracket, \\ \hat{m} \llbracket m \wedge x \rrbracket \end{array} \right\} \vee \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket \neg x \rrbracket, \\ \hat{x} \llbracket 0 \rrbracket, \\ \hat{y} \llbracket y \wedge \neg x \rrbracket, \\ \hat{z} \llbracket y \wedge \neg x \rrbracket, \\ \hat{m} \llbracket y \wedge \neg x \rrbracket \end{array} \right\}$$

$$= \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket 1 \rrbracket, \\ \hat{z} \llbracket y \wedge (f_{reg} \vee \neg x) \rrbracket, \\ \hat{m} \llbracket (m \wedge x) \vee (y \wedge \neg x) \rrbracket \end{array} \right\}$$

Now **ComposeTS** terminates (case JPC) returning the composition of $\widehat{C}_{jpc}$ with $\widehat{C}_{1a}$ i.e., with the operations before JPC.

$$\begin{aligned} \widehat{C}_1 &= \widehat{C}_{jp} \circledcirc \widehat{C}_{1a} \\ &= \left\{ \hat{f}_{reg} \llbracket 1 \rrbracket, \hat{z} \llbracket y \wedge (f_{reg} \vee \neg x) \rrbracket, \hat{m} \llbracket (m \wedge x) \vee (y \wedge \neg x) \rrbracket \right\} \circledcirc \{ \hat{f}_{reg} \llbracket x \rrbracket \} \\ &= \left\{ \hat{f}_{reg} \llbracket 1 \rrbracket, \hat{z} \llbracket y \rrbracket, \hat{m} \llbracket (m \wedge x) \vee (y \wedge \neg x) \rrbracket \right\} \end{aligned}$$

After **ComposeTS**$(1,1)$ has returned, LOOP should continue, but this step down for-loop has reached final index 1, so APLCTRANS terminates and transfer set $\widehat{C}_1$ contains t-assignments describing the operations of whole APLC program.

APLCTRANS algorithm implementation, surveyed on page 89, gives this result written in the notation suitable for text files:

```
Composed in 0.18 seconds.
Result={ @f[(1.(1.x))+(1.!(1.x))],
         m[(m.(1.x))+(((1.y)).!(1.x))],
         z[((((1.x).y)).(1.x))+(((1.y)).!(1.x))] }
Minimized in 0.00 seconds.
MinResult={ @f[1], m[x.m+!x.y], z[y] }
```

Symbol '@f' stands for f_{reg}. The prefix '@' is reserved code for APLC machine internal variables to distinguish them from all names used in programs.

3.4.2 Discussion

In this subsection, we consider properties of APLCTRANS, namely its termination and complexity issues.

Termination

APLCTRANS algorithm contains two for-loops that will certainly terminate after reaching final indexes. The recursive calls of **ComposeTS** are invoked only in the case of composing JPC instructions. If JPC has $\widehat{C}_{ilist,j}$ transfer set then we invoke **ComposeTS**$(\mathit{ilist}, j+1)$, which proceeds the tail of current block *ilist*

$$\widehat{C}_{\mathit{ilist},j+1} :: \widehat{C}_{\mathit{ilist},j+2} :: \ldots :: \widehat{C}_{\mathit{ilist},m_{\mathit{ilist}}}$$

The main **ComposeTS**, which has invoked the recursive call, terminates immediately after obtaining its result. Therefore, *each $\widehat{C}_{\mathit{ilist},j}$ is evaluated only one times regardless of a number of recursive calls.* After rewriting the composition of Jpc with the aid of some stack for internal variables of **ComposeTS**, the recursion could be removed completely, but the algorithm will be less readable.

So APLCTRANS always terminates after providing βn compositions of transfer sets where n stands for the number of instructions and β is a finite constant that expresses unknown number of calls and jumps in the program. They add at most 3 additional compositional operations, so $\beta \leq 3$ for all APLC programs.

Memory requirements

APLCTRANS needs to store only the transfer sets of evaluated blocks and requires few temporary variables. Critical elements are only transfer sets, which could be very huge structures under certain conditions discussed in the next part.

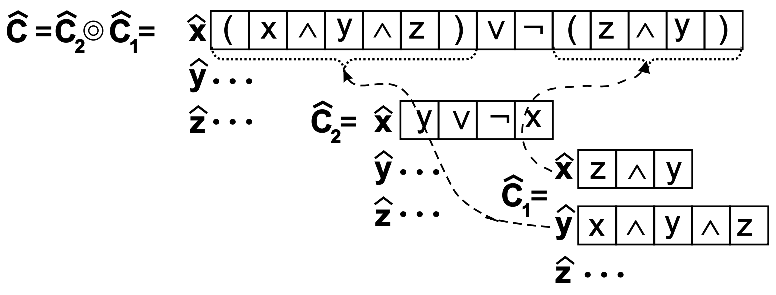
Moreover, Example 3.11 has shown that the compositions create boolean formulas with easily minimizable combinations, such as this expression $1 \wedge x$, $x \wedge (\neg x \wedge y)$. Reducing these formulas will have certainly significant influence on the growth of stored data.

Therefore, we consider for each of the suggested implementations of the transfer sets not only requirements for memory, but also appropriateness for a run-time minimization of boolean expressions.

The first implementation stores the t-assignments as numeric arrays. Each variable symbol is coded as a unique number with the size of byte or 16 bit integer according to requisite number of variables. [14] Reserved numeric codes are also assigned to all boolean operations and parentheses.

We have derived in the proof of Proposition 3.8 that $\circledcirc$ composition replaces all variables by the strings of a rightmost operand. If we employ this principle then the composition of our integer arrays means replacing each variable code by the content of the array, to which the code points, and enclosing the inserted part in parentheses (i.e., the codes assigned to them) as outlined in Figure 3.3. The boolean operations with transfer sets can be also implemented by proper joining contents of the arrays.

[14]APLCTRANS will probably never deal with the number of variables that would overflow 16 bit integer. PLC programs have usually hundreds (or at most thousands) of variables. Moreover, any program with more than 2^{16} variables will be probably too complex and far beyond limits of todays formal methods.



$\hat{C} = \hat{C}_2 \circledcirc \hat{C}_1 =$
$\hat{x}$ (x ∧ y ∧ z) ∨ ¬ (z ∧ y)
$\hat{y} \cdots$
$\hat{z} \cdots$
$\hat{C}_2 =$ $\hat{x}$ y ∨ ¬ x
$\hat{y} \cdots$
$\hat{z} \cdots$
$\hat{C}_1 =$ $\hat{x}$ z ∧ y
$\hat{y}$ x ∧ y ∧ z
$\hat{z} \cdots$

Figure 3.3: Implementation of ⊚ as Arrays

The anticipated advantages of the arrays are simple implementation and stream operations. Each array is read or written only in the direction from its first element to the end, which allows storing its whole content to a disc file. If advance operating systems services are employed for files that are momentary only read (as transfer sets of already calculated blocks) [15] then APLCTRANS becomes limited technically only by available space on computer disc, or disk arrays, respectively. Transfer sets that store the result of a momentary performed composition can be only kept in memory to simplify minimizations.

The arrays have many drawbacks. Each composition accesses nearly all stored data and could gradually increase their sizes. Employing throughout minimization is necessary and its algorithm becomes the main component responsible for APLCTRANS computational complexity.

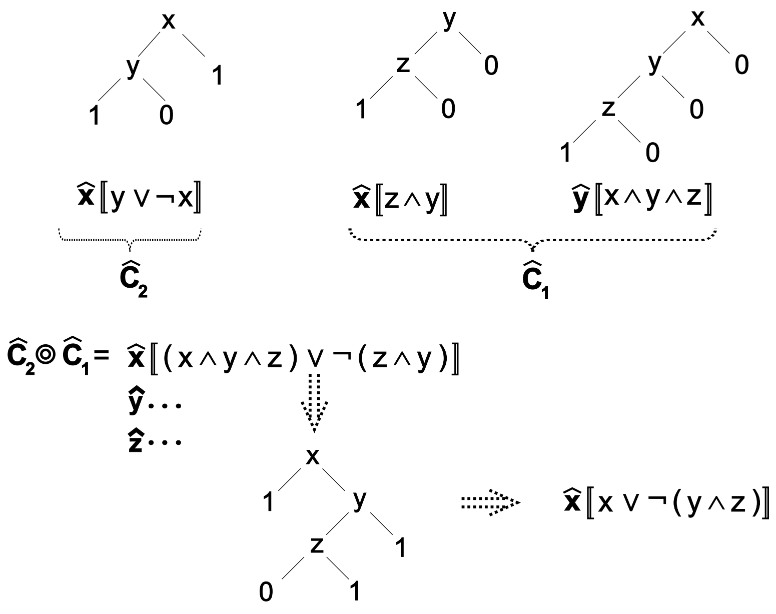
Binary decision diagrams (BDD) allow better minimizations. This problem si intensely studied in the literature, for example [HD93, OCVSV98]. The tutorial introduction to BDD was published by Andersen in [And97]. Software packages are also available — their overview is in [YBO$^+$98].

But significant disadvantage of such approach is missing possibility of direct BDD compositions. Binary decision diagrams contain the references to variables in every node and concurrent substitutions require replacing variables by new expressions.

It is doubtful whether could be ever constructed an algorithm capable of providing efficient concurrent substitutions directly with BDD. Thus, BDDs must be always converted to expressions and new BDD is built from the results of their compositions, as shown in Figure 3.4.

Moreover, the worst cases of BDD sizes are exponential. The exact equations for various BDD types are presented in Gröpl's thesis [Grö99]. Even if we suppose that BDDs will have no exponential size, then the construction of BDD has high complexity.

[15]For example, 32-bit Windows offer mapping files into system address space without previous committing any allocation of memory. The files become integrated part of system swap file, which speeds up access to them [Šus99, And94]



$\hat{x}[\![y \vee \neg x]\!]$
$\hat{x}[\![z \wedge y]\!]$
$\hat{y}[\![x \wedge y \wedge z]\!]$
$\hat{C}_2$
$\hat{C}_1$
$\hat{C}_2 \circledcirc \hat{C}_1 = \hat{x}[\![(x \wedge y \wedge z) \vee \neg(z \wedge y)]\!]$
$\hat{y}\ldots$
$\hat{z}\ldots$
$\hat{x}[\![x \vee \neg(y \wedge z)]\!]$

Figure 3.4: Implementation of Transfer Sets by BDDs

Proposition 3.13 *A BDD of a function $f : \mathcal{B}^v \to \mathcal{B}$ represented by n minterms can be constructed in $O(n^2 v^2 \log n)$ operations.*

Proof: The proof is performed by BDD construction and it is presented in [OCVSV98, page 3] □

Therefore, BDD are not suitable for run-time implementation of transfer sets, but they may be used for the representation of final results.

The best structure for transfer sets are probably Binary expression diagrams (BEDs). In contrast to formulas, BEDs allow sharing terms and the same logical subexpressions are not repeated, which reduce memory requirements, as shown in Figure 3.5. BEDs was described by Andersen and Hulgaard in several publications [AH97, HWA97, HWA99].

If boolean subexpressions are shared among all t-assignments of one transfer set, then the compositions mean replacing all variable references in leftmost $\circledcirc$ operand by links to rightmost operand.

The major advantage of BEDs is their tree structure that allow providing simple minimizations, as shown in the bottom of Figure 3.5. Moreover, BEDs have indeed good complexity properties, which we will discuss below.

Complexity

Firstly, it must be conceded that APLCTRANS complexity can be answer only by practical experience with PLC programs. Theoretical solutions are

$$\widehat{C}_2 \quad \left\{\begin{array}{l} \hat{x}[\![y\vee\neg x]\!] \\ \hat{y}[\![x\vee y]\!] \\ \hat{z}[\![z]\!] \end{array}\right\}$$

$$\circledcirc$$

$$\widehat{C}_1 \quad \left\{\begin{array}{l} \hat{x}[\![z\wedge y]\!] \\ \hat{y}[\![x\wedge y\wedge z]\!] \\ \hat{z}[\![z\wedge y]\!] \end{array}\right\}$$

$$=$$

$$\left\{\begin{array}{l} \hat{x}[\![(x\wedge y\wedge z)\vee\neg(z\wedge y)]\!] \\ \hat{y}[\![(z\wedge y)\vee(x\wedge y\wedge z)]\!] \\ \hat{z}[\![z\wedge y]\!] \end{array}\right\}$$

$$=$$

$$\left\{\begin{array}{l} \hat{x}[\![x\vee\neg(z\wedge y)]\!] \\ \hat{y}[\![(z\wedge y)\vee x]\!] \\ \hat{z}[\![z\wedge y]\!] \end{array}\right\}$$

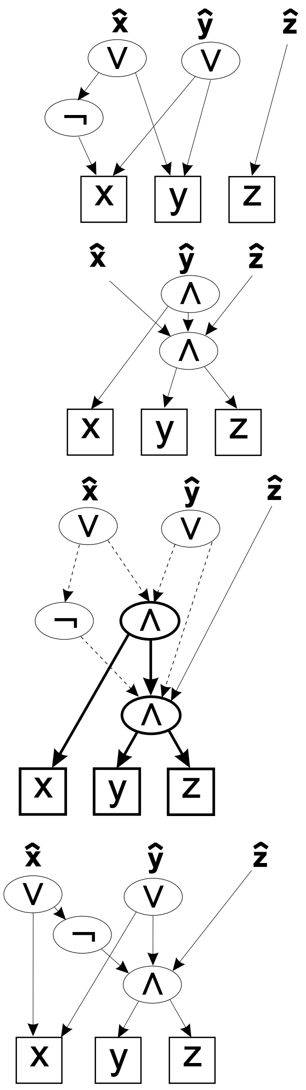


Figure 3.5: Implementation of ◎ by BEDs

in many ways unsatisfactory because many logical functions have exponential complexity and we easily create a contradiction.

Overview of main results in the complexity of logic function was published by Boppana and Sipser, who also discussed well-known Muller's theorem [BS90, page 9].

Proposition 3.14 (Muller) *Almost every boolean function of n variables requires circuits of size $O(2^n/n)$.*

Muller's theorem shows the existence of boolean functions with exponential circuit complexity. Because formulas are special types of circuits whose gates have outputs connected only to one gate, this conclusion is also valid for boolean expressions.

On one hand, special types of boolean functions exist, for which had been proved polynomial complexity, [16] but the most of 2^{2^n} possible boolean of n variables will have an exponential complexity.

Therefore, the fundamental question is about *whether a boolean function with exponential complexity really appears in transfer sets during APLCTRANS conversion of a PLC program.*

The transfer sets are the main factor that determines APLCTRANS complexity. The number of the operations with transfer sets needed for composing a program depends on the number of instructions linearly, which could suggest $O(n)$ complexity The worst case APLC program (in terms of APLC steps) presented in Table 3.4 (see page 48) requires only n simple compositions

$$\left(\{\ \hat{f}_{reg}[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E\right) \;=\; \widehat{\prod}_{i=n}^{1<} \left(\{\ \hat{f}_{reg}[\![1]\!]\ \} \cup \widehat{\Theta}(0)^E\right)$$

even if the program scan takes $2^n - 1$ instruction steps.

More complex situation appears for the second worst case in Table 3.5. Let us write $\widehat{C}_k$ for the transfer set of k block, where k is number of blocks, each containing m instructions, so the program length is $n = k * m$ instructions. Then, the transfer set of whole program equals to

$$\widehat{C_1} \;=\; \underbrace{\widehat{\prod}_{i=m}^{1<}\left(\cdots\widehat{\prod}_{i=m}^{1<}\left(\widehat{\prod}_{i=m}^{1<}\widehat{C}_k\right)\cdots\right)}_{k-1\,\times\,\prod} \tag{3.47}$$

$$= \widehat{\prod}_{i=1}^{m^{k-1}} \widehat{C}_k \tag{3.48}$$

Though APLCTRANS will calculate $\widehat{C_1}$ after $\beta(k-1)m = \beta(n-m)$ compositions of $\widehat{C}_k$ employing Equation (3.47) where β was introduced on page

[16]For example, lower bound size $O(n^{2.38})$ was proved for monotone formulas for multiplication of Boolean matrices. (Monotone formula use only $\vee$ and $\wedge$ operations.) [BS90].

83, the amount of memory allocated for transfer sets could extremely grow together with the time for manipulations with them, which could increase the computational complexity of the whole algorithm.

We will narrow considerations only to BED, which are the most effective structures. Unfortunately, Andersen and Hulgaard did not make exact propositions about worst case BEDs size, with the exception of misguiding notice: "BEDs can represent any Boolean circuit in linear space" [AH97, page 1]. More precise formulation contains Williams's thesis [Wil00, Observation 20 on page 14]:

> BED size is linear in the size of the formula. BDD size is exponential.

This is correct because BEDs represent software variants of circuits and they are more close to them than boolean formulas. If we reverse the directions of the arrows in a BED graph we see that BED recommends logical circuits and differs from them only in two main properties:

- BEDs are acyclic graphs, but general logical circuits can contain loops, and
- BEDs allow nodes with two sub elements, but circuits usually have gates with more inputs.

Therefore, the most of circuit properties also hold for BEDs. Many logical functions exist whose representation will cause BED data explosion. But Williams's observation bypasses this complexity problem by the implication: *If we had represented some logical function by a finite circuit then this function could not belong among the family of functions with exponential sizes.*

Using this observation, which Williams has supported by practical experiments, we present a similar hypothesis for all PLCs, whose program can be expressed as ladder diagram or function block diagram (see page 22), because the both graphical programming languages are very close to circuits. Because APLC language allows entering complex expressions, we will relate the size of a transfer set to the size of APLC source code, not to the number of instructions.

Hypothesis 3.1 *Almost every PLC program, which is expressible either in ladder diagram or function block diagram and convertible to APLC program, can be represented as one transfer set whose size is* linear in the size of APLC source code*.*

Critique: Even if the most of PLC program could have such representations under favorable conditions, no assumption ensures that this optimal state will be really achieved. Thus we should rather expect a polynomial growth of transfer set sizes at the best case. $\boxed{?}$

This hypothesis together with its critique is also supported by our practical experiments. The conditions for its validity or the exact equations of polynomial data growth are a matter for further study.

Experimental Results

To analyze the behavior of APLCTRANS, we have written its test version, whose main purpose was to reveal critical problems and gave concepts for improvements. This goal was achieved and a new version of APLCTRANS is now in development.

The test version of APLCTRANS was implemented with the aid of byte arrays. We did not use BED, because there is only one available package for them [17] and its extension to the composition of transfer sets has appeared too complex. The package should be probably partially rewritten.

Byte arrays are capable to distinguish at most 238 programs variables. The remaining codes (up to 256) are reserved for a termination code, boolean operations, f_{reg}, and the evaluation stack variables. The boolean operations $\equiv$ and $\not\equiv$ were not implemented yet. Practical tests show that the number of variables was underestimated and many PLC programs need more variables to express their temporary operations. But 16 bit coding of variables should satisfy all requirements.

APLCTRANS was designed as a multi pass compiler. If some of the passed described below is not needed then it is skipped. Therefore, the input can be either APLC instructions or import a source code of PLC.

The first pass processes PLC program, which is converted to APLC language. PLC variable names are replaced by symbols according to supplied PLC symbolic database file, which constitutes necessary input data for APLCTRANS operation together with PLC source code listing (see example in Figure 3.2 on page 80).

The second pass creates the list of all used variables and labels and divides APLC program into blocks, which are arranged in such order that jumps and calls do not invoke backward addresses. If such ordering is not possible, APLCTRANS terminates with announcing backward call error.

The instructions in the blocks are optimizes during third pass by recognizing frequent sequences of APLC instructions and joining them together. For example, "AND b_1; AND b_2;" is replaced by one equivalent instruction AND $b_1 \wedge b_2$. We have expected that this optimization will accelerate the compositions, but it has yielded only negligible effect.

The forth pass evaluates the compositions. They are usually calculated unexpectedly fast as shown in Figures 3.6 and 3.7. The figure depicts compositional time and the sizes of transfer sets that were obtained by composing

[17] BED package was written by P. Williams, H. Hulgaard, and H. Andersen and it is available at URL http://www.it-c.dk/research/bed/

a fragment of PLC program, that controls a big round coal store in Taiwan harbor.

To obtain the number of variables suitable for our test version of APLCTRANS, we could process only the part, which deals with system diagnostic, error detection, sequential control of drives, coal scrapper and conveyer. Because PLC program was known, the author of this thesis cooperates on its writing, we could make this extraction to preserve the fragment functional. It contained 6004 codes of PLC 5 processor that were compiled into 8479 APLC instructions divided in 86 compositional blocks.

In contrast to the hypothesis presented above, we will relate all measured data to the number of APLC instructions, because PLC 5 processor codes were all convertible into APLC instructions, which have at most a single variable in their expressions. Therefore, the number of instruction depends linear on the size of APLC program.

The upper diagram shows the relationship between the count of processed APLC instructions and composition time, when no minimizations are applied and APLCTRANS performs only raw compositions of transfer sets. The lower diagram depicts the characteristics of the same program, but composed with a minimization option switched to providing the reductions of transfer sets after each composition.

The left side of the diagram shows the composition of the last block, which contained sequential controls. The size curve has middle part nearly linear due to composing many relatively simple diagnostic and error detections subroutines. They were called from introductory part of the program and their composition caused the fast growth of the size around 8000th instruction. [18] Perpendicular drop at the end expresses releasing the transfer sets of all blocks when APLCTRANS has finished.

The final sizes of transfer sets (13.7 kB or 132.3 kB) correspond to coded expressions, in which each variable is stored as one byte. Replacing all codes by variable names enlarged minimized formula to 235 expressions with total size 109 kB. [19] This is probably more that abilities of any checker tool and so presented results have meaning only as a case study of APLCTRANS behavior.

Surprisingly, APLCTRANS shows unexpected speed of the compositions and nearly linear time for non minimized operation. [20] Thorough checks

[18]Notice, that the numbers of instructions in the figure do not strictly represent their order in the program because LOOPFN of **ComposeTS()** function reads each block in forward direction, but main part of APLCTRANS composes the blocks in backward direction. The first instruction of the first compositional block is near the right side of the diagram but not at its end (precisely, the first instruction was numbered as 8459 in the diagram).

[19]109 kB is the third of LaTeX 2_ε text this thesis approximately. Non minimized composition had 313 kB after decoding. It was expanded in lesser ratio than the minimized formula because it contained many parentheses.

[20]The tests were performed on 400 MHz PC with 256 MB of RAM.

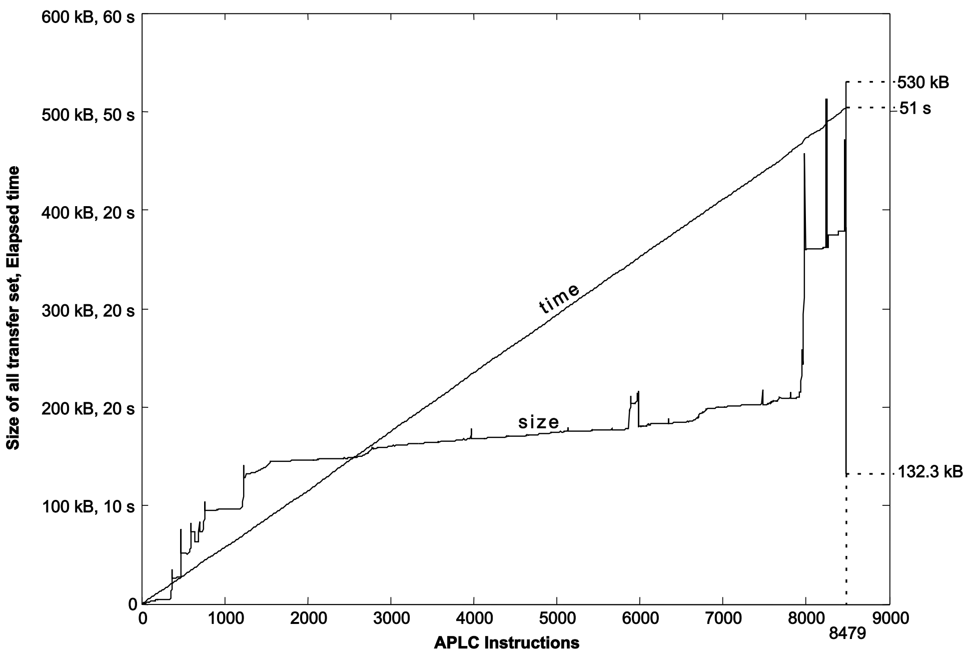


Figure 3.6: Example of APLCTRANS Operation without Minimization

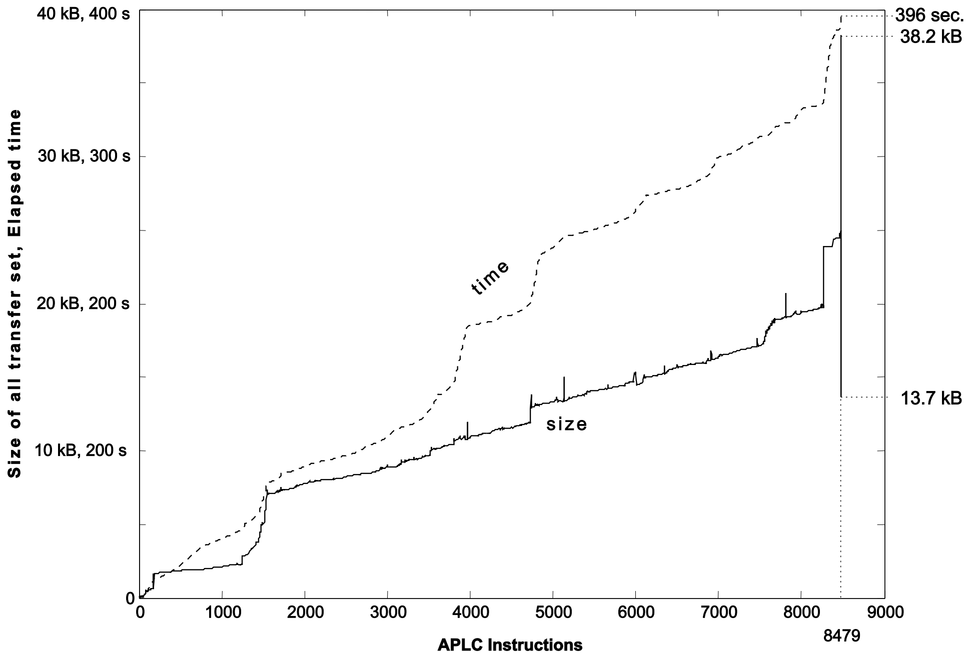


Figure 3.7: Example APLCTRANS Operation with Minimization

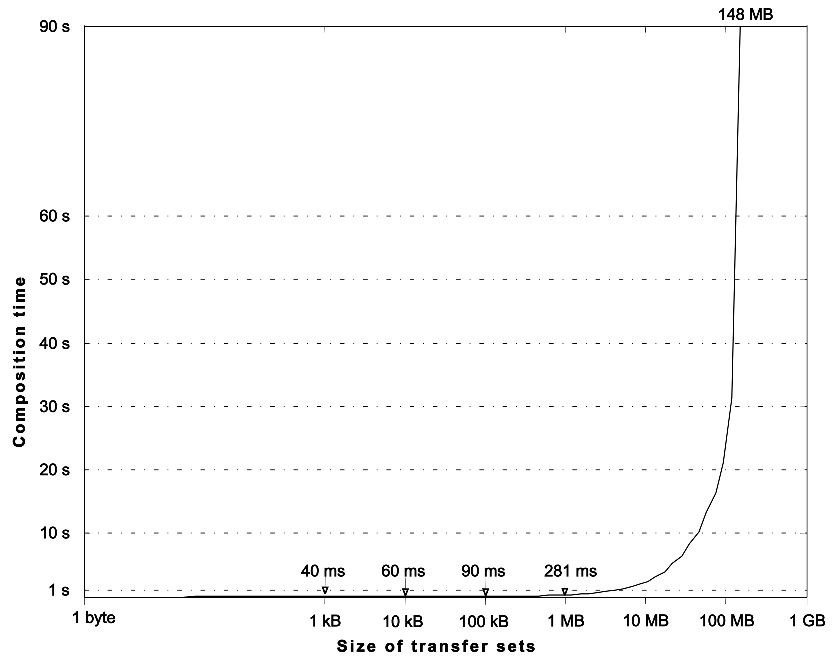


Figure 3.8: Dependency of Composition Time on the Size of Transfer Sets

have revealed that the compositions loops are delayed mainly by run-time translation of pre compiled APLC coded into transfer sets and the manipulations with small amounts of data are minor factors.

To verify this fact, simple APLC program containing repeated operations $x = x \wedge y; y = y \wedge x;$ was tested. When APLCTRANS minimization option is off, the size of all transfer sets increased by a fourth after each composition approximately. The measurement of individual composition times has yielded the diagram in Figure 3.8, which supports the assumption that the compositions are slowed down more only in the case of megabyte sizes of transfer sets.

When the minimization was switched off, APLCTRANS processed 65 instructions before 'out of memory' exception has occurred. If the minimization was allowed after each composition, 65 instructions took 40 miliseconds and the maximum size of all transfer sets did not exceed 21 bytes. This result could suggest that minimizations will give better results even if it is sometimes calculated in longer time. But this conclusion is invalid in general.

One of PLC programs that were available for tests violates this assumption as shown in Table 3.10 and Figure 3.10 (see pages 95 and 96). They summarize the results of the test performed with 19 fragments extracted from PLC programs, that were borrowed for this research by courtesy of firms SPEL, Blumenbecker Prague, and SIDAT.

All programs control industrial technologies and they were written by different programmers. Unfortunately, we could process only programs for PLC 5, because the import modules for another PLCs have not been finished yet, and we could compose only fragments, since every PLC program contains some unconvertible parts.

Thus the sample is not so weighty as we would wish, and no statistically significant relation can be deduced. But the results indicate that some weak relation usually exists between the number of APLC instructions, composing time and the size of final transfer set.

The program were ordered according to total number of APLC instruction. The table presents the size of the final transfer set with coded expressions and time required for the composition when minimize option is switched off to demonstrate pure APLCTRANS operations.

The exception are only lines emphasized by bold font. They represent one program that was composed with minimized option on (10a) and off (10b) and it has produced contradict.

To explain it, we created worst case example as xor-network defined by the equations:

$$\begin{aligned} x &= (x_0 \wedge x_1) \vee (\neg x_0 \wedge \neg x_1) \\ x_r &= (x_{r.0} \wedge x_{r.1}) \vee (\neg x_{r.0} \wedge \neg x_{r.1}) \text{ for } i < n \end{aligned}$$

where $r \in \{0,1\}*$ and $r.0$ or $r.1$ is concatenation operation (see page 139) and n represents the depth of xor network.

Because APLCTRANS compiler assigns $\wedge$ higher priority than $\vee$ automatically, the corresponding APLC program (if we emphasize r by subscripting for better readability) is

```
            Init;
            Load $x_{00000} . x_{00001} + !x_{00000} . !x_{00001}$;Store $x_{0000}$;
...⇑...     it continues up to 5 XOR levels;
            Load $x_{000} . x_{001} + !x_{000} . !x_{001}$; Store $x_{00}$;
            Load $x_{010} . x_{011} + !x_{010} . !x_{011}$; Store $x_{01}$;
            Load $x_{100} . x_{101} + !x_{100} . !x_{101}$; Store $x_{10}$;
            Load $x_{110} . x_{111} + !x_{110} . !x_{111}$; Store $x_{11}$;
            Load $x_{00} . x_{01} + !x_{00} . !x_{01}$; Store $x_0$;
            Load $x_{10} . x_{11} + !x_{10} . !x_{11}$; Store $x_1$;
            Load $x_0 . x_1 + !x_0 . !x_1$;
            Store $x$;
            End;
```

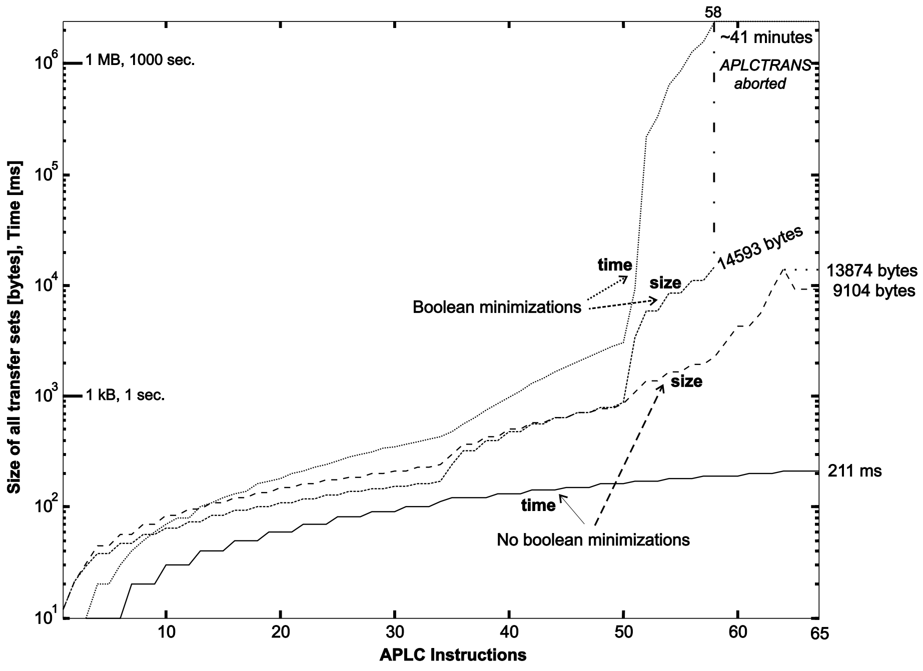


Figure 3.9: Composition of XOR Network

The diagram of APLCTRANS behavior is presented in Figure 3.9. When the minimization was switched on, APLCTRANS was aborted after 41 minutes after processing only 58 instructions. Xor network above is non minimizable. Simple minimization algorithm based on boolean laws did not recognize this fact and tried expand formulas to $2^{2^{(n-1)}}$ and-minterms. In the contrast, APLCTRANS without minimization has finished after 201 ms. Notice that xor-network would yield a transfer set whose size will be linear in the size of the program source code, in both cases, if transfer set would have been implemented as BEDs.

If we combine xor-network with the program used for creating Figure 3.8 we obtain the program, which will show exponential behavior with minimization option either on or off as the result 10a and 10b in Table 3.10. But this is mainly caused by low quality of our simple minimization algorithm.

Using some heuristics combined with a better minimization tool, for example by SIS [SSL+92] or by BOOM [FH01a, FH01b, FH01c, FH01d], could probably solve this problem and APLCTRANS would show lower computational complexity in more cases. Further research is required on proper methods for minimizing transfer sets.

By tests, we have shown that APLCTRANS performs surprisingly well. Although, there are always possibilities of APLCTRANS exponential behav-

Program	**Number of APLC instr.**	**Size of final transfer set [kB]**	**Composing time [s]**
1	438	1.7	2.2
2	565	2.6	2.9
3	914	8.9	4.9
4	1056	3.7	5.8
5	1077	4.9	6.3
6	1166	6	6.3
7	1575	6.7	8.9
8	1662	6.7	9.4
9	1670	156.5	11.8
min. **10a**	**1990**	**29300**	**954**
no min. **10b**		**77436**	**1736**
11	2312	16.7	14
12	2376	11.5	14.3
13	2825	14.5	16
14	3199	22	19.4
15	3572	24.3	22
16	3581	11.6	20.8
17	4296	13.6	24.5
18	4429	10.6	25.6
19	8479	132.3	51

Table 3.10: Tested APLC Programs *(see also Figure 3.10)*

ior either in time or in space, we believe that the algorithm will operate with polynomial complexity of some lower order for *almost every* PLC program. Further experiments will certainly show how this assumption is correct.

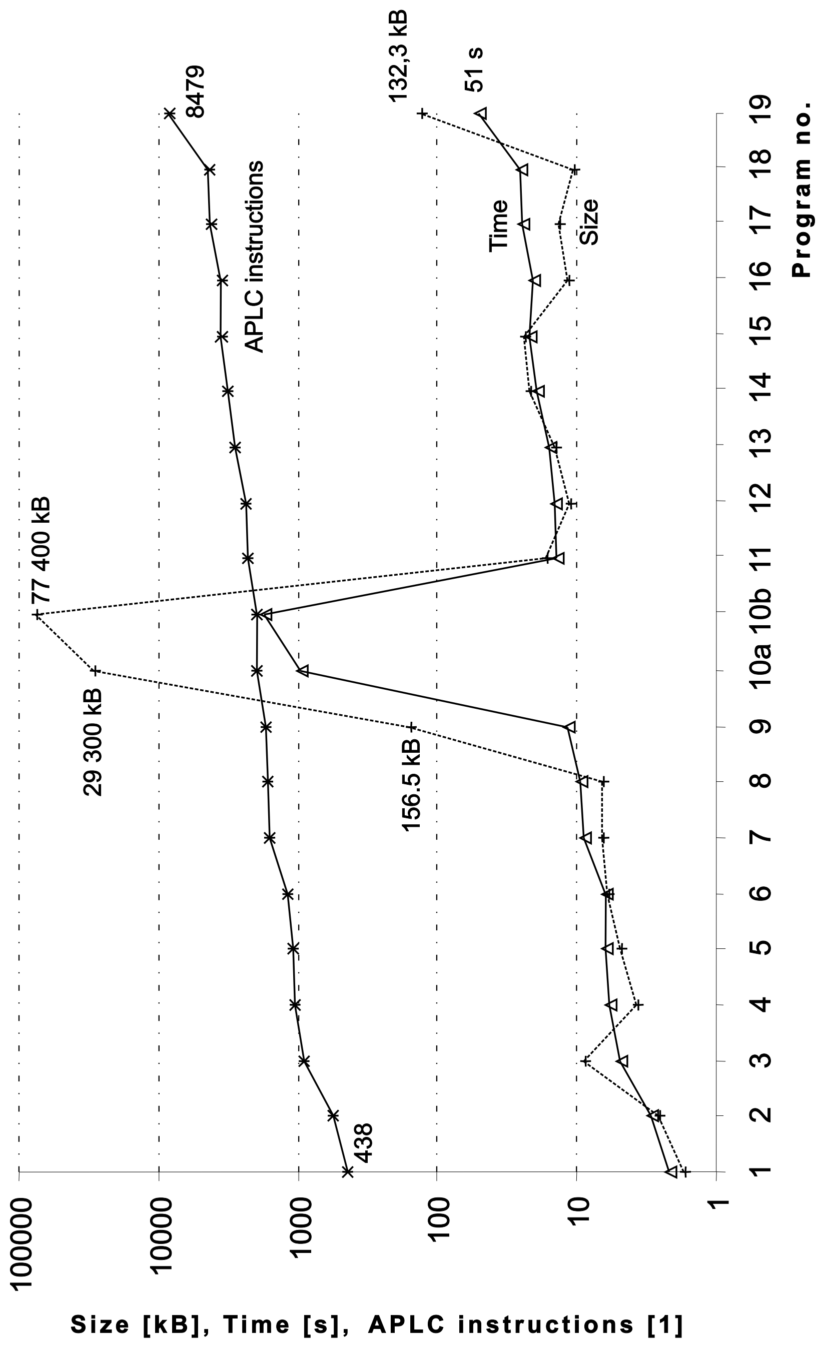


Figure 3.10: Tested APLC Programs

Chapter 4

Abstract PLC Applications

This chapter presents the outline of several verification methods of PLCs based on APLCTRANS. To demonstrate its wide usage, we also overview its easy modifications for expressing PLCs as a timed or mode automaton.

4.1 Automaton Generated by APLC Program

To improve automaton generated by binary PLC, Definition 3.3 (on page 33), we begin by putting together previous conclusions. Considering properties of transfer sets generated by APLCTRANS we will attempt to achieve a better assumption for the number of automaton states than theoretical value $2^{|\Omega|+|V|}$ i.e., all possible contents of PLC internal memory S specified by Equation 3.43 (see page 73).

First, we define the intersection of a transfer set with a subset of S storage:

Definition 4.1 *Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be any transfer set on S variables and U any subset of S, then we define:*

$$\widehat{X} \mathbin{\widehat{\cap}} U \;\overset{df}{=}\; \left\{ \hat{x} \in \widehat{X} \mid \mathrm{co}(\hat{x}) \in U \right\}$$

and we extend Definition 3.13 on page 54 to transfer sets:

Definition 4.2 *Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be any transfer set defined on storage S then*

$$\begin{aligned} \mathrm{co}(\widehat{X}) &= \left\{ x \in S \mid x \mathbin{\widehat{\in}} \widehat{X} \right\} = \bigcup_{i=1}^{|\widehat{X}|} \mathrm{co}(\hat{x}_i) \quad \textit{where } \hat{x}_i \in \widehat{X} \\ \mathrm{dom}(\widehat{X}) &= \bigcup_{i=1}^{|\widehat{X}|} \mathrm{dom}(\hat{x}_i) \quad \textit{where } \hat{x}_i \in \widehat{X} \end{aligned} \tag{4.1}$$

The sets $\mathrm{co}(\widehat{X})$ *and* $\mathrm{dom}(\widehat{X})$ *are called a* codomain *and a* domain *of* $\widehat{X}$.

In other words, the domain (or the codomain) of a transfer set is the union of domains (or codomains) of all its t-assignments. Now, we must narrow their definition to variables that are important for *BPLC*.

Definition 4.3 *Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be any transfer set defined on BPLC storage S, which has as its subset $(\Omega \cup V) \subseteq S$ then we define*

$$\begin{aligned}
\widehat{\mathrm{co}_P}(\widehat{X}) &= \left\{ \hat{x} \in \left((\widehat{X} \,\widehat{\cap}\, (\Omega \cup V)) {\downarrow} \right) \,\middle|\, \hat{x} \,\widehat{\not\equiv}\, \hat{x}\,[\![\mathit{0}]\!] \;\wedge\; \hat{x} \,\widehat{\not\equiv}\, \hat{x}\,[\![\mathit{1}]\!] \right\} \\
\mathrm{co}_P(\widehat{X}) &= \mathrm{co}\left(\widehat{\mathrm{co}_P}(\widehat{X}) \right) = \left\{ x \in S \mid x \,\widehat{\in}\, \widehat{\mathrm{co}_P}(\widehat{X}) \right\} \\
\widehat{\mathrm{dom}_P}(\widehat{X}) &= \left\{ \hat{x} \in (\widehat{X} \uparrow S) \mid \mathrm{co}(\hat{x}) \in \mathrm{dom}(\widehat{X}) \right\}
\end{aligned}$$

The transfer set $\widehat{\mathrm{dom}_P}(\widehat{X})$ is called a PLC domain *of $\widehat{X}$, and the transfer set $\widehat{\mathrm{co}_P}(\widehat{X})$ and the set $\mathrm{co}_P(\widehat{X})$ are called* PLC codomains of $\widehat{X}$.

PLC codomain of $\widehat{X}$, i.e. $\widehat{\mathrm{co}_P}(\widehat{X})$, contains only t-assignments, whose values depend on another variables i.e, all constants and canonical t-assignments are excluded. Its corresponding codomain set $\mathrm{co}_P(\widehat{X})$ consists of variables whose t-assignments are in $\widehat{\mathrm{co}_P}(\widehat{X})$.

In contrast, PLC domain of $\widehat{X}$, i.e. $\widehat{\mathrm{dom}_P}(\widehat{X})$, contains only t-assignments for variables, on which depends $\mathrm{co}_P(\widehat{X})$. These t-assignments are taken from $\widehat{X} \uparrow S$, which means: if a variable has no t-assignments in $\widehat{X}$, i.e. $x_i \,\widehat{\notin}\, \widehat{X}$, then its canonical t-assignment is substituted.

Notice that $\widehat{\mathrm{co}_P}(\widehat{X})$ definition is based only on $\Omega \cup V$ variables, because *we must exclude all elements corresponding to inputs Σ — they are stored in PLC memory and therefore it is possible to change their value by* Store *instruction.*

However, such assignments seem to be meaningless, they are sometimes used for changing input or output addresses. These operations are PLC specialties are require informal explanation.

Suppose occurring a malfunction of input r_1 and output w_1 on some I/O module, e.g. due to an overvoltage. The module operates correctly, with the exception of damaged r_1 input and w_1 output. Instead of expensive replacing the whole module, input r_1 and output w_1 can be reconnected to some free ports on this one or another I/O module. Let us write r_2 and w_2 for their new addresses.

The PLC has been repaired electrically, but a correction of software is also necessary. A find-replace edition of its source code could help, but with high probability of some incorrect or forgotten changes. The better way represents changing PLC memory by copying r_2 to r_1 before the program scan, and w_1 to w_2 after the scan.

```
NewStart:     LOAD    r2;
              STORE   r1;
              JS      OldProgram;
              LOAD    w1;
              STORE   w2;
              END;
OldProgram:   ...
```

In this case, the associativity of $\circledcirc$ composition has an interesting consequence. Let $\widehat{C}_{new}$ and $\widehat{C}_{old}$ be the transfer sets of the new and old program evaluated by APLCTRANS algorithm, then $r_2, w_2 \notin \mathrm{dom}(\widehat{C}_{old})$ implies $r_1, w_2 \notin \mathrm{dom}(\widehat{C}_{new})$.

This follows directly from the associativity of $\circledcirc$ composition (see proof of Proposition 3.8 on page 66). If *OldProgram* does not uses variables r_2 and w_2 then the two instructions inserted before the call of *OldProgram* ensure replacing all occurrences of r_1 in $\widehat{C}_{old}$ by r_2 and two appended instructions replace only one occurrence of w_2 in the domain of new program by w_1, so *the old input r_1 and new output w_2 will not participate in the transfer set of the new program.*

In other words, $\circledcirc$ *compositions assures not appearing auxiliary variables, which are assigned before their usage, in the domain of the transfer set of whole $AP \in AProgram$.* But they remain in its codomain and their proper recognition requires an extra information about their usage — that is unsolvable by mere semantic analysis.

We can removed only possible Σ part of the auxiliary variables. *PLC input image is always rewritten during following input scan by new read data, therefore no information can be passed to a next program scan in the input image.* If $x \in \Sigma$ exists such that $x \in \mathrm{co}_P(\widehat{X})$ then x is surely a auxiliary variable that has no influence of its state. Thus we have narrowed PLC codomain by the intersection with $\Omega \cup V$.

Practical application of transfer sets need their evaluation to elements of $\alpha()$, introduced in Definition 3.1 (see page 31). The evaluation could be performed by $\circledcirc$ operator, but such approach requires many additional definitions.

Therefore, we rather convert a transfer set to a mapping. We have not defined an ordering for the transfer sets, such supplement will be certainly possible, but not too homogeneous with $\uparrow$ and $\downarrow$ operators. We will prefer a mapping of transfer sets based on PLC codomains and domains introduced above.

If $\widehat{X}$ is any transfer set on a storage S then its domain $R = \mathrm{dom}(\widehat{X})$ and PLC codomain $P = \mathrm{co}_P(\widehat{X})$ are finite sets of boolean variables, i.e $P, R \subset S \subset \mathcal{B}$. We may create their ordering, $P = \{p_i \in P \mid i \in I, |I| = |P|\}$, as required by $\alpha()$ definition. Any $p^\alpha \in \alpha(P)$ is represented by a string with length $|I|$ bits, i.e. the permutations of 0 or 1 values that correspond to P variables. We do the same for R.

The orderings of P and R allow a definition of $\psi : \alpha(R) \to \alpha(P)$ as a mapping $\{0,1\}^{|R|} \to \{0,1\}^{|P|}$ which can be constructed as the union of $|P|$ mappings $\{0,1\}^{|R|} \to \{0,1\}$ specified by:

$$f_j(x_i) : [\![\mathit{bexp}_i]\!] \to \{0,1\} \quad \text{where } i \in |I| = |P| \tag{4.2}$$

where $\mathit{bexp}_i \in \mathit{Bexp}^+$ is the boolean expression that belongs to $\hat{x}_i[\![\mathit{bexp}_i]\!] \in \widehat{X}$ t-assignment for $x_i \in P$. We employed the fact that $\mathrm{dom}(\hat{x}_i[\![\mathit{bexp}_i]\!]) \subseteq R$ and bexp_i expression itself will map any $r_j \in \alpha(R)$ to $\{0,1\}$, $j \in I, |I| = |R|$ because R is the domain of $\widehat{X}$.

Definition 4.4 *Let S be a storage S, which has as its subset $(\Omega \cup V) \subseteq S$, and $\widehat{X}$ be any transfer set on S with non-empty finite PLC domain $R = \mathrm{dom}(\widehat{X})$ and non-empty finite PLC codomain $\widehat{P} = \widehat{\mathrm{co}_P}(\widehat{X})$. Let us write $P = \mathrm{co}_P(\widehat{X}) = \mathrm{co}(\widehat{P})$.*

If some orderings of R and P are given then we define $\mathrm{map}_W(\widehat{X})$ mapping by the following construction:

$$\begin{aligned}
\mathrm{map}_W(\widehat{X}) &\overset{df}{=} \psi : \alpha(R) \to \alpha(P) \\
\textit{where} \qquad & \psi : \alpha(R) \to \alpha(P) \overset{df}{=} \bigcup_{i=1}^{|\widehat{P}|} \left(f_i(x_i) : \alpha(R) \to \{0,1\} \right) \\
& \textit{where } f_i(x_i) : \alpha(R) \to \{0,1\} \overset{df}{=} f_i(x_i) : [\![\mathit{bexp}_i]\!] \to \{0,1\} \\
& \qquad \textit{for all } \hat{x}_i[\![\mathit{bexp}_i]\!] \in \widehat{P} \\
& \qquad\qquad \textit{such that } \mathrm{co}(\hat{x}_i[\![\mathit{bexp}_i]\!]) = x_i \textit{ and } x_i \in P
\end{aligned}$$

Example 4.1

Practical application of the definition above is relatively easy. We demonstrate its usage on the result of Example 3.11 (see page 79):

$$\widehat{C_1} \;=\; \left\{ \hat{f}_{reg}\,[\![1]\!]\,, \hat{z}\,[\![y]\!]\,, \hat{m}\,[\![(m \wedge x) \vee (y \wedge \neg x)]\!] \right\}$$

that was defined for the storage S with BPLC subsets:

$\Sigma = \{x, y\}$, $\Omega = \{z\}$, and $V = \{m\}$.

We first calculate $\widehat{C_1}$ domain as the set of variables, on which its t-assignments depends:

$$\mathrm{dom}(\widehat{C_1}) \;=\; \{m, x, y\}$$

then we easily derive PLC domain from it by finding corresponding t-assignments in $\widehat{C_1} \uparrow S$:

$$\widehat{\mathrm{dom}_P}(\widehat{C_1}) \;=\; \{\hat{m}\,[\![(m \wedge x) \vee (y \wedge \neg x)]\!]\,, \hat{x}\,[\![x]\!]\,, \hat{y}\,[\![y]\!]\}$$

To obtain PLC codomain of $\widehat{C_1}$, we select (from $\widehat{C_1}$) only t-assignments for variables in $\Omega \cup V$ a we omit all canonical and constant t-assignments. In our example, we only delete $\hat{f}_{reg}\,[\![1]\!]$ from $\widehat{C_1}$ which yields:

$$\begin{aligned}\widehat{\mathrm{co}_P}(\widehat{C_1}) &= \{\hat{m}\,[\![(m \wedge x) \vee (y \wedge \neg x)]\!]\,, \hat{z}\,[\![y]\!]\}\\ \mathrm{co}_P(\widehat{C_1}) &= \mathrm{co}\left(\widehat{\mathrm{co}_P}(\widehat{C_1})\right) = \{m, z\}\end{aligned}$$

then we construct $\mathrm{map}_P(\widehat{X})$ as $\langle m, z\rangle = \psi(\langle m, x, y\rangle)$ which is given by the union of two mappings:

$$f_1(m) : [\![(m \wedge x) \vee (y \wedge \neg x)]\!] \rightarrow \{0, 1\} \qquad \text{and} \qquad f_2(z) : [\![y]\!] \rightarrow \{0, 1\}$$

The mapping can be easily rewritten into definition languages of many checker tools.

Notice that the transfer sets of whole APLC program, $\widehat{C}$ in the example above, has *its domain that does not contain internal variables of APLC machine*, because an evaluation stack and f_{reg} are initialized by the terminal END instruction, therefore the rightmost composition always substitutes them by constants. Internal variables of APLC machine may be certainly present in $\mathrm{co}(\widehat{C})$, but never in $\mathrm{dom}(\widehat{C})$. It follows from the properties of $\circledcirc$ compositions discussed in the previous section.

Proposition 4.1 (Automaton generated by *AProgram*)
Let $AL \in AProgram$ be any APLC program defined on Σ inputs, Ω outputs, and V internal variables. If $\widehat{C}$ transfer set of AL was created by APLC-TRANS, then the automaton generated by AL is the tuple

$$\mathbf{M}(\widehat{C}) \overset{df}{=} \langle X, Y, Q, \delta, q_0, \omega\rangle \tag{4.3}$$

where is

$X = \alpha\Big(\mathrm{dom}(\widehat{C}) \cap \Sigma\Big)$ - *input alphabet,*

$Y = \alpha\Big(\mathrm{co}(\widehat{C}) \cap \Omega\Big)$ - *output alphabet,*

$Q = \alpha\Big(\mathrm{co}_P(\widehat{C}) \cap \mathrm{dom}(\widehat{C})\Big)$ - *set of the states,*

$q_0 \in Q$ - *an initial state of the automaton,*

$\delta = \mathrm{map}_P\Big(\widehat{\mathrm{co}_P}(\widehat{C}) \cap \widehat{\mathrm{dom}_P}(\widehat{C})\Big)$ - *transition function $\delta : Q \times X \rightarrow Q$,*

$y = \mathrm{map}_P\Big(\widehat{C} \mathbin{\widehat{\cap}} \Omega\Big)$ - *output function $\omega : Q \times X \rightarrow Y$*

Proof: Both δ and ω are defined by the t-assignments that are converted by map_P into normal boolean functions (mappings). The equations for X, Y, and ω follow directly from properties of automata and PLCs, which read input data only into Σ and send to peripherals only data from Ω. Y and

ω are not restricted to $\mathrm{co}_P(\widehat{C})$ because the definition above contains no assumption about the usage of Ω variables.

We will only prove the expression for δ transition function. Q is derived from it. Associativity $\circledcirc$ ensures that auxiliary variables do not appear in domain. Since transition function δ of automata is defined as a mapping $\delta : Q \times X \to Q$ and δ only depends on such t-assignments that change the variables belonging to the intersection of the domain and codomain. Let us write $Z = \left(\mathrm{co}_P(\widehat{C}) \cap \mathrm{dom}(\widehat{C})\right)$ for the intersection, in this proof only.

We proceed by contradiction. Let $v \in \Omega \cup V$ be a variable such that v has any influence on some $q \in Z$ and $v \notin Z$.

Therefore v must be missing either in the domain or in the codomain. If $v \notin \mathrm{dom}(\widehat{C})$ then v has no influence on any t-assignment in $\widehat{X}$ including $\hat{v}$ t-assignment of itself. In this case, either $\hat{v}$ value depends on another variables or $\hat{v} \in \widehat{\mathcal{E}}^S$, so $\hat{v}$ has been removed by $\downarrow$ compression, In both situations, v cannot store any state information.

Finally, we consider the case $v \notin \mathrm{co}_P(\widehat{C})$, but it means that v value is never changed and it immediately follows that v is constant.

We reach a contradiction to our assumption that v has any influence on a value on $q \in Z$. $\square$

Notice that $\mathbf{M}(\widehat{C})$ belong to Mealy's family in general, because ω is created from $\widehat{\mathrm{co}_P}(\widehat{C})$ and $(\mathrm{dom}(\widehat{\mathrm{co}_P}(\widehat{C})) \cap \Sigma$ could be (possibly) a non empty set. Also, $\mathrm{co}_P(\widehat{C}) \cap \mathrm{dom}(\widehat{C}) \cap \Omega$ need not be empty and a part of states can be stored in output variables.

The fundamental proposition of the theory of automata says that any automaton of Mealy's family can be converted to automaton of Moore's family, and vice versa. The both automata have an equivalent behavior but Mealy's form of an automaton has always the number of its states less or equal than corresponding Moore's form of that automaton.

Therefore, Proposition 4.1 has improved Definition 3.3 (see page 33) and presented much better model for *BPLC* introduced in Definition 3.2 (see page 32).

The previous results allow presenting the fundamental proposition of this subsection:

Proposition 4.2 *Any program of a binary PLC can be modeled by an automaton of Mealy's family if and only if it is expressible as an APLC program $AL \in AProgram$.*

Proof: We have shown in this chapter that APLC program is always convertible into an automaton of Mealy's family. We prove the reverse implication by a construction of APLC program from an automaton. The structure of final APLC program is shown in Table 4.1 on page 105.

Let $M = \langle X, Y, Q, \delta, q_0, \omega\rangle$ be an Mealy's automaton with X input alphabet, Y output alphabet, Q states, q^0 an initial state, transition function $\delta : Q \times X \rightarrow Q$, and output function $\omega : Q \times X \rightarrow Y$.

We define an ordering of X, Y, and Q and we find least integer numbers that satisfy the constraints:

$$n_X \geq \log_2 |X|,\ n_Y \geq \log_2 |Y|,\ \text{and } n_Q \geq \log_2 |Q|$$

Using them we construct the sets of codes

$$\begin{aligned} X_\alpha &= \alpha(I) \text{ where } |I| = n_X \\ Y_\alpha &= \alpha(I) \text{ where } |I| = n_Y \\ Q_\alpha &= \alpha(I) \text{ where } |I| = n_Q \end{aligned}$$

Each set is a language defined over $\{0,1\}^*$ and its strings are expressible as a concatenation of $\{0,1\}$ alphabet elements. We convert the elements to values of binary variables. If $x^\alpha \in X^\alpha$ is a string then we express its values with the aid of $X_v = \{x_1, x_2, \ \ldots \ x_{n_X}\}$ variables with $x_\alpha = x'_1.x'_2. \ \ldots \ x'_{n_X}$ values from $\{0,1\}$. The strings of Y_α and Q_α are expressed similarly to Y_v and Q_v.

We have obtained the coding sets whose cardinalities are equal or greater than cardinalities of X, Y, and Q and therefore we may assign unique element (i.e. a unique string of 0 and 1) to each member of X, Y, and Q.

Now we add binary variables defined in X_v, Y_v, and Q_v, which allows to convert δ to the set of n_Q boolean functions of $n_Q + n_X$ variables $Q_v \cup X_v$. We also perform similar conversion of ω to the set of n_Y boolean functions of $n_Q + n_X$ variables $Q_v \cup X_v$. These functions define two mappings:

$$\begin{aligned} F_\delta :&\ \ \{0,1\}^{n_Q+n_X} \rightarrow \{0,1\}^{n_Q} \\ F_\omega :&\ \ \{0,1\}^{n_Q+n_X} \rightarrow \{0,1\}^{n_Y} \end{aligned}$$

The both mappings assign different outputs and so we join them to one set F of $n_Q + n_Y$ boolean functions of $n_Q + n_X$ variables $Q_v \cup X_v$.

We proceed to writing APLC program. To assure proper evaluation of all expressions, we must copy all variables in Q_v to new set $\underline{Q_v}$ of temporary variables. We denote its members by underlining.

Replacing references to variables in F expressions by these underlined variables yields $\underline{F}$ set of $n_Q + n_Y$ boolean functions of $n_Q + n_X$ variables $\underline{Q_v} \cup X_v$, which differ from F only by readdressing of variables.

Now all expression strictly operate with the same data and evaluated results do not interfere with inputs. Since associativity properties of $\circledcirc$ remove these temporary copies, they do not increase the number of states.

To perform initialization, we add new input *fs* to X_v, which will signal the first program scan after starting program. Manufactured PLCs provide this initialization by different ways, but all methods are convertible to invocation of an initialization subroutine when first-scan signal appears. [1]

When fs will be active we invoke copying variables values in $q^0 = q^0_1. \dots q^0_{n_Q}$ to Q_v.

$\square$

We have shown that a binary PLC can be expressed as an automaton of Mealy's family if and only if it is possible to rewrite its program into APLC language. Mealy's automata also include automata of Moore's family as a subset and are always coverable to them. Therefore, the proposition holds for the both types of automata.

Hypothesis 4.1 *Any PLC program can be modeled by an automaton of Mealy's family iff it is expressible as an APLC program $AL \in AProgram$.*

Critique: We have shown in Proposition 4.2 that this hold for binary PLC. To prove the same for any PLC, we should first create a conversion mechanism for all PLC programs, which is very complex task due to wide range of PLC types. Therefore the hypothesis above will probably remains only a hypothesis forever. $\boxed{?}$

[1]Some PLCs has this signal built in as read only system bit that behaves as an additional input, eg. PLC 5 or SLC 500.

$$
\begin{array}{lll}
\mathit{Start} & : & \left.\begin{array}{ll} \textsc{Init} & \\ \textsc{Load} & \mathit{fs} \\ \textsc{Jsc} & \mathit{Initialize} \end{array}\right\} q_0 \\
 & & \left.\begin{array}{ll} \textsc{Load} & q_1 \\ \textsc{Store} & \underline{q_1} \\ \ldots & \\ \textsc{Load} & q_{n_Q} \\ \textsc{Store} & \underline{q_{n_Q}} \end{array}\right\} \text{Copying data} \\
 & & \left.\begin{array}{ll} \textsc{Load} & \underline{\mathit{bexp}_{q_1}} \\ \textsc{Store} & q_1 \\ \ldots & \\ \textsc{Load} & \underline{\mathit{bexp}_{q_{n_Q}}} \\ \textsc{Store} & q_{n_Q} \end{array}\right\} \text{Evaluate } \delta \\
 & & \left.\begin{array}{ll} \textsc{Load} & \underline{\mathit{bexp}_{y_1}} \\ \textsc{Store} & y_1 \\ \ldots & \\ \textsc{Load} & \underline{\mathit{bexp}_{y_{n_Y}}} \\ \textsc{Store} & y_{n_Y} \\ \textsc{End} & \end{array}\right\} \text{Evaluate } \omega \\
\mathit{Initialize} & : & \left.\begin{array}{ll} \textsc{Load} & q_1^0 \\ \textsc{Store} & q_1 \\ \ldots & \\ \textsc{Load} & q_{n_Q}^0 \\ \textsc{Store} & q_{n_Q} \\ \textsc{End} & \end{array}\right\} \text{Initialize memory}
\end{array}
$$

Table 4.1: APLC Program of Mealy's Automaton

4.2 Decomposition of APLC Automaton

The decomposition formalizes the approach that is often used for debugging and troubleshooting of PLCs. When programmers and operators search for errors, they intuitively simplify a program by reducing states taken in question, which generally consists of ignoring some aspects of the program involved that have no influence to an analyzed function.

Since many systems have their natural division into components or modules, proving properties of individual parts could imply some specifications about the entire system. These techniques are a part of the compositional verification [McM00].

This approach can reduce a complex problem $P \models \Phi$ (P satisfies Φ) to a set of smaller problems $P'_i \models \Phi'_i, i = 1, \ldots, n$. If we denote $O(x)$ computational complexity of a task x then we try to find out such decomposition that satisfies:

$$\sum_{i=1}^{n} O(P'_i \models \Phi'_i) < O(P \models \Phi) \tag{4.4}$$

$$(P'_1 \models \Phi'_1) \wedge (P'_2 \models \Phi'_2) \wedge \ldots \wedge (P'_n \models \Phi'_n) \Rightarrow (P \models \Phi) \tag{4.5}$$

There are several ways of dividing systems to smaller units, but the most powerful is parallel decomposition if it exists, because many properties of individual parallel systems are preserved for whole process.

Proposition 4.3 (Parallel composition)
Let $\mathbf{M}_1$ *and* $\mathbf{M}_2$ *be automata generated by two APLC programs that were defined on the same storage* $S = \{\Sigma, V, \Omega\}$*:*

$$\mathbf{M}_i \;=\; \left\langle X_i, Y_i, Q_i,\ \delta_i,\ q_0^i,\ \omega_1 \right\rangle \text{ for } i = 1, 2$$

such that $X_i = \alpha(\Sigma_i)$ *for* $i = 1, 2$ *and* $\Sigma_1 \cup \Sigma_2 \subseteq \Sigma$*. Suppose that two mappings from* $X = \alpha\left(\Sigma_1 \cup \Sigma_2\right)$ *to* X_1 *and* X_2 *can be constructed such that they map each* x *to corresponding* x_1 *and* x_2*. Then, the parallel composition is defined as*

$$\begin{array}{rrcl} & \mathbf{M} = \mathbf{M}_1 || \mathbf{M}_2 & \overset{df}{=} & \langle X, Y_1 \times Y_2, Q_1 \times Q_2,\ \delta,\ q_0,\ \omega \rangle \\ \textit{where} & q_0 & = & \left\langle q_0^1, q_0^2 \right\rangle \\ & \delta\left(\langle q_1, q_2 \rangle, x\right) & = & \langle \delta_1(q_1, x_1), \delta_2(q_2, x_2) \rangle \\ & \omega\left(\langle q_1, q_2 \rangle, x\right) & = & \langle \omega_1(q_1, x_1), \omega_1(q_2, x_2) \rangle \end{array}$$

for all $q_1 \in Q_1, q_2 \in Q_2$ *and* $x \in X$*.*

If $\Sigma_1 \cap \Sigma_2 = \emptyset$ then the definition corresponds to a usual definition of *cartesian product without synchronization* [BBF$^+$01, page 14], in which $X = X_1 \times X_2$. Otherwise, it is *state parallel composition* with shared inputs.

Many publications dealing with automata present the definitions of parallel compositions. Demlová and Koubek [DK90] also dealt with the reverse problem — the decomposition of a Mealy's automaton to two parallel ones. Their work is probably unique.

Using their conclusions we have created the algorithm for decomposing of PLCs programs published in [Šus02]. But this work has been now overcome by APLCTRANS, which finds out the decomposition in more cases than old algorithm. Therefore, we outline only such definitions and propositions from [Šus02] and [DK90], which we will refer in the next parts to.

Definition 4.5 *Given an automaton* $\mathbf{M} = \langle X, Y, Q,\ \delta,\ q_0,\ \omega\rangle$ *of Mealy's family. Equivalence* ρ *on set* Q *of states is called an* automaton congruence of automaton $\mathbf{M}$*, if* $\langle \delta(q,x), \delta(r,x)\rangle \in \rho$ *holds for any input* $x \in X$ *and for all pairs of states* $q, r \in Q$ *such that* $\langle q, r\rangle \in \rho$*.*

Definition 4.6 *Let* $\mathbf{M} = \langle X, Y, Q,\ \delta,\ q_0,\ \omega\rangle$ *be an automaton of Mealy's family, and given* ρ *automaton congruence of automaton* $\mathbf{M}$*, then*

$$\mathbf{M}/\rho \overset{df}{=} \left\langle X, Y' = Q/\rho\ \times X, Q' = Q/\rho, \delta', \omega' \right\rangle \tag{4.6}$$

is called state factor automaton, if the following holds

$$\begin{aligned} \delta'\left(\widetilde{\rho}(q), x\right) &= \widetilde{\rho}\left(\delta(r,x)\right) \\ \omega'\left(\widetilde{\rho}(q), x\right) &= \langle \widetilde{\rho}(q), x\rangle \end{aligned}$$

where $\widetilde{\rho}$ *denotes factor set (see page 137).*

Proposition 4.4 *An automaton* $\mathbf{M}$ *has a non trivial state parallel decomposition if and only if two non trivial separating automaton congruence of automaton exist. Let us write* ρ *and* τ *for them, then* $\mathbf{M}/\rho$ *and* $\mathbf{M}/\tau$ *are non trivial state parallel decompositions of automaton* $\mathbf{M}$*.*

Proof: The proof is proceeded by the construction of the automaton and uses terminology that was not presented above. Full proof was published in [DK90, page 174] □

Automaton congruence of automaton determines the completeness of state parallel decomposition and allows its construction. But known Kong's algorithm for searching automaton congruences of automaton [DK90, page 264] has complexity $O(|Q|^2)$ where $|Q|$ denotes the number of states. Its direct usage for automata generated by an APLC programs means a methodical anachronism — we generate an automaton that requires a laborious reduction. Decomposing APLC program offers more methodical approach. We only need to specify when APLC program will generate the automaton, in which an automaton congruence of automaton exists.

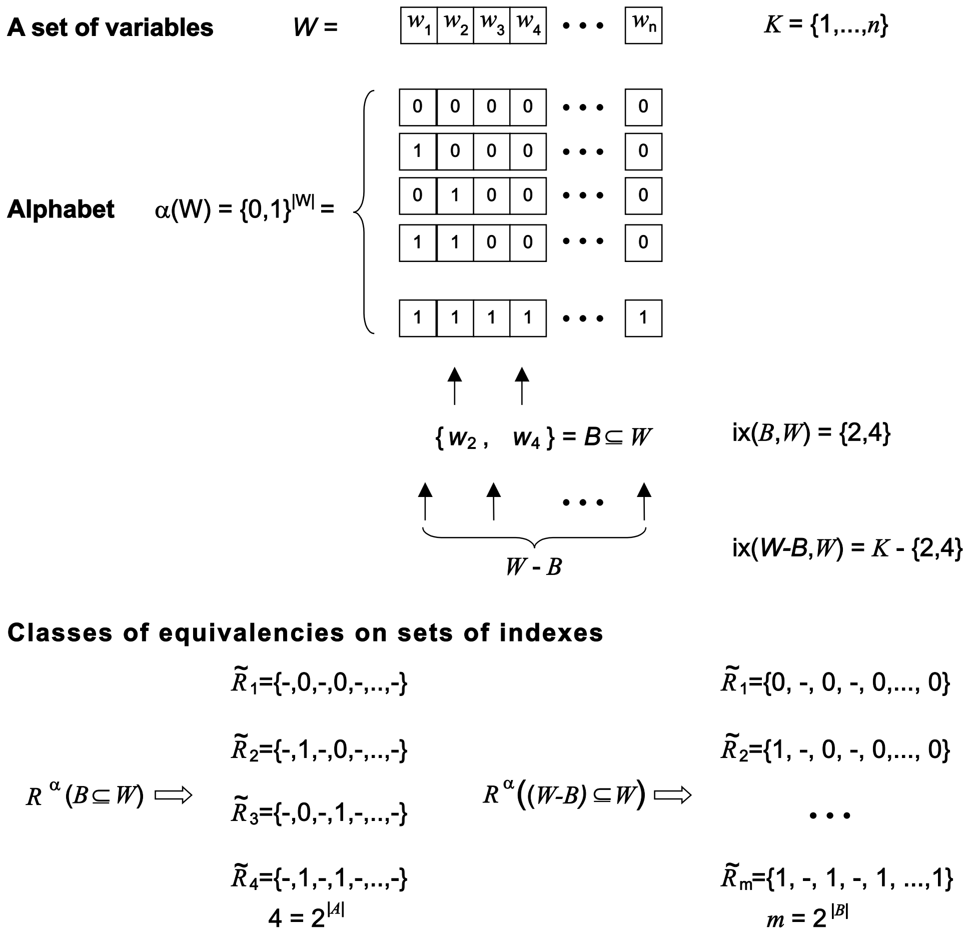


Figure 4.1: Separating Equivalences on Sets of Indexes

Definition 4.7 (Equivalence on set of indexes)
Let W be e non empty finite ordered set of binary variables and let us write K for its index set $K = I, |I| = |W|$. We denote by $W[i]$ variable $w \in W$ with index $i \in K$. Let B be a subset of W then we define $\mathrm{ix}(B \subseteq W)$ *(read the set of B indexes on W) and the equivalence on this set as the following:*

$$\begin{aligned} \mathrm{ix}(B \subseteq W) &\overset{df}{=} \{i \mid W[i] \in B\} \\ \overset{L}{\equiv} &\overset{df}{=} \{\langle x, y\rangle \in W \times W \mid x[i] = y[i] \textit{ for all } i \in L = \mathrm{ix}(B \subseteq W)\} \end{aligned}$$

Verifications of PLCs often lead to some equivalences on sets of indexes. For example, if we should verify CTL safety property $\mathrm{AG}\neg(x \wedge y)$ i.e., both x and y will never go true, then we do not check a reachability of one state. In actuality, if we denote state variables of a PLC by W, then this property requires checking non reachability of states defined by the set:

$$\{q \in \alpha(W) \mid (q[i] = 1) \wedge (q[j] = 1) \text{ for all } i, j \in \text{ix}\,(\{x, y\} \subseteq W)\}$$

Therefore, many verification properties should be transformed into equivalences on set of indexes. To show their connection to an automaton congruences of automaton, we need next proposition, whose construction is outlined in Figure 4.1 for a better readability.

Proposition 4.5 *Let W be a non empty ordered set of $n = |W|$ binary variables with given non empty subset $B \subseteq W$. Let us write $R^\alpha(B \subseteq W)$ for the equivalence on set $\alpha(B)$ defined as:*

$$R^\alpha(B \subseteq W) \overset{df}{=} \left\{ \langle x, y\rangle \in \alpha(W) \times \alpha(W) \mid x \overset{L}{\equiv} y \textit{ where } L = \text{ix}(B \subseteq W) \right\}$$

then two equivalences $R^\alpha(B \subseteq W)$ and $R^\alpha((W - B) \subseteq W)$ create a system of separating equivalences on set $\alpha(W)$.

Proof: The equivalences on the set of indexes are certainly reflexive, symmetric and transitive, because all these properties directly follow from comparing their elements by $=$ operator.

We must only show that $R^\alpha(B \subseteq W) \cap R^\alpha((W - B) \subseteq W) = \Delta_W$ to prove separating property i.e., the intersection of equivalences is trivial equivalence (see page 137). We will proceed by a contradiction. Let $u, v \in \alpha(W)$ be two variables $u \neq v$ such that

$$\langle u, v\rangle \in R^\alpha(B \subseteq W) \quad \text{and} \quad \langle u, v\rangle \in R^\alpha\left((W - B) \subseteq W\right)$$

It means that they satisfy the conditions of the both equivalences. Let us write $I_1 = \text{ix}(B \subseteq W)$ and $I_2 = \text{ix}((W - B) \subseteq W)$, then

$$u \overset{I_1}{\equiv} v \quad \text{and} \quad u \overset{I_2}{\equiv} v$$

Applying equality $I = I_1 \cup I_2 = \text{ix}(W \subseteq W)$ where $|I| = |W|$ yields $u \overset{I}{\equiv} v$. It means $u = v$ and we have a contradiction.

Now we show that the intersection is exactly Δ_W, but this follows from the fact that the both equivalences contain pairs $\langle x, x\rangle$ and all such pairs, because $x \overset{J}{\equiv} x$ holds for any $x \in \alpha(W)$ and any arbitrary non empty subset $J \subseteq I, |I| = |W|$. □

Using separating properties and automaton congruence of automata we determine the soundness of parallel decomposition of automaton generated by APLC program.

Proposition 4.6 (Soundness) *Let $AL \in AProgram$ be APLC program defined on storage $S = \{\Sigma, V, \Omega\}$ and given its transfer set $\widehat{C}$ created by APLCTRANS. If two transfer sets $\widehat{C}_1$ and $\widehat{C}_2$ exist such that they satisfy the following:*

$$\begin{aligned}
\emptyset &= \mathrm{co}_P(\widehat{C}_1) \cap \mathrm{co}_P(\widehat{C}_2) && (4.7)\\
\mathrm{co}_P(\widehat{C}) &= \mathrm{co}_P(\widehat{C}_1) \cup \mathrm{co}_P(\widehat{C}_2) && (4.8)\\
\emptyset &= \mathrm{co}_P(\widehat{C}_1) \cap \mathrm{dom}(\widehat{C}_2) && (4.9)\\
\emptyset &= \mathrm{co}_P(\widehat{C}_2) \cap \mathrm{dom}(\widehat{C}_1) && (4.10)
\end{aligned}$$

then $\mathbf{M}(\widehat{C}) = \mathbf{M}_1(\widehat{C}_1) \mid\mid \mathbf{M}_2(\widehat{C}_2)$.

Proof: First, we show the existence of the system of separating equivalences for $\mathbf{M}(\widehat{C})$. We utilize the fact that $\mathrm{dom}(\widehat{C}) = \mathrm{dom}(\widehat{C}_1) \cup \mathrm{dom}(\widehat{C}_2)$, which follow directly from Definitions 4.2 and 4.3 of transfer set domains and Equation 4.8.

Equations 4.7 and 4.8 determine that $\widehat{C}$ is separated into two parts, which can share only t-assignments belonging to Σ, because they are not included in PLC codomains. But we have shown in Proposition 4.1 that t-assignments of Σ always correspond to temporary variables and have no influence to the states.

Proposition 4.1 has shown that the states equal to $Q_i = \alpha(W_i)$ where $W_i = \mathrm{co}_P(\widehat{C}_i) \cap \mathrm{dom}(\widehat{C}_i)$ for $i = 1, 2$. It certainly holds that $W \supseteq (W_1 \cup W_2)$ where $W = \mathrm{co}_P(\widehat{C}) \cap \mathrm{dom}(\widehat{C})$. We prove that $W = W_1 \cup W_2$ by contradiction.

Let $x \in W$ be a variable such that $x \notin W_1$ and $x \notin W_2$. But x should satisfy $x \in \mathrm{co}_P(\widehat{C}_i)$ for either $i = 1$ or $i = 2$ necessarily, otherwise x does not belong to Q defined in Proposition 4.1, because $\mathrm{co}_P(\widehat{C})$ was divided into two parts (Equations 4.7 and 4.8), hence x must belong to one of them.

Suppose that $x \in \mathrm{co}_P(\widehat{C}_1)$, otherwise we swap the indexes. Equation 4.9 and assumption $x \notin W_1$ yields $x \notin \mathrm{dom}(\widehat{C}_2)$. But W is defined as the intersection of $\widehat{C}$ domain and codomain. To satisfy $x \in W$, it should hold that $x \in \mathrm{dom}(\widehat{C}_1)$. It yields that $x \in W_1$ and we have a contradiction.

Thus $W = W_1 \cup W_2$. Therefore, $R_1^\alpha(W_1 \subseteq W)$ and $R_2^\alpha(W_2 \subseteq W)$ are the system of separating equivalences on $\alpha(W)$.

We show that this system is the automaton congruence of automaton $\mathbf{M}(\widehat{C}) = \mathbf{M}_1(\widehat{C}_1)||\mathbf{M}_2(\widehat{C}_2)$ by its construction. Parallel composition of $\mathbf{M}_1||\mathbf{M}_2$ has the set of states $Q = Q_1 \times Q_2$, where $Q_i = \alpha(W_i)$ for $i = 1, 2$. Because $W_1 \cap W_2 = \emptyset$, then $Q = \alpha(W_1) \times \alpha(W_2)$ corresponds to the union of W_1 and W_2, which yields $Q = \alpha(W_1 \cup W_2)$.

Equation 4.7 yields the similar conclusions for output alphabet Y and also for output function ω which is based on Y, as the following

$$\begin{aligned}
Y = Y_1 \times Y_2 &= \alpha\Big(\mathrm{co}(\widehat{C}_1 \cup \widehat{C}_2) \cap \Omega\Big)\\
\omega &= \mathrm{map}_P\Big(\Big(\widehat{C}_1 \cup \widehat{C}_2\Big) \mathbin{\widehat{\cap}} \Omega\Big)
\end{aligned}$$

Equations 4.9 and 4.10 allow to express the transition function as

$$\begin{aligned}
\delta\left(\langle q_1, q_2\rangle, x\right) &= \langle \delta_1(q_1, x_1), \delta_2(q_2, x_2)\rangle \\
&= \left\langle \begin{array}{c} \operatorname{map}_P\Big(\widehat{\operatorname{co}_P}(\widehat{C}_1) \cap \widehat{\operatorname{dom}_P}(\widehat{C}_1)\Big), \\ \operatorname{map}_P\Big(\widehat{\operatorname{co}_P}(\widehat{C}_2) \cap \widehat{\operatorname{dom}_P}(\widehat{C}_2)\Big) \end{array} \right\rangle \\
&= \operatorname{map}_P\Big(\widehat{\operatorname{co}_P}(\widehat{C}_1 \cup \widehat{C}_2) \cap \widehat{\operatorname{dom}_P}(\widehat{C}_1 \cup \widehat{C}_2)\Big)
\end{aligned}$$

and we must show that δ satisfies

$$\langle \delta(q, x), \delta(r, x)\rangle \in R_i^\alpha(W_i \subseteq W) \text{ where } i = 1, 2$$

for any input $x \in X$ and for all states $q, r \in Q$ such that the pair of the states $\langle q, r\rangle \in R_i^\alpha(W_i \subseteq W)$.

In other words, all states belonging to the same equivalence class either should remain in it or they should be moved to an identical equivalence class simultaneously.

But this holds obviously because Equations 4.9 and 4.10 assure cross-independency of $\widehat{C}_1$ and $\widehat{C}_2$. T-assignments of $\widehat{C}_1$ do not depend on $\operatorname{co}_P(\widehat{C}_2)$, thus they only change W_1 variables according to inputs and therefore all states belonging to one equivalence class $R_1^\alpha(W_1 \subseteq W)$ are moved by transition function to identical equivalence class concurrently.

The same property satisfies $\widehat{C}_2$ and $R_2^\alpha(W_2 \subseteq W)$. Thus, two non trivial separating automaton congruency of automaton exist, if $\mathbf{M}(\widehat{C})$ is state parallel composition of $\mathbf{M}_1(\widehat{C}_1)$ and $\mathbf{M}_2(\widehat{C}_2)$. $\square$

4.2.1 Algorithm for Parallel Decomposition

Inputs: Let $\widehat{C}$ be the transfer set of APLC program defined of storage $S = \{\Sigma, V, \Omega\}$.

Initilization: We first calculate $\widehat{\operatorname{co}_P}(\widehat{C})$ (see Definition 4.2 on page 97), then we create the following sets:

$$\begin{aligned} D_i &= \{\operatorname{co}(\hat{w}_i)\} \cup (\operatorname{dom}(\hat{w}_i) - \Sigma) \\ & \qquad \text{for all } \hat{w}_i \in \widehat{\operatorname{co}_P}(\widehat{C}),\ i \in I,\ |I| = |\widehat{\operatorname{co}_P}(\widehat{C})| \end{aligned} \tag{4.11}$$

Each set D_i contains one codomain variable and the domain of its corresponding t-assignment, from which inputs are subtracted, because their sharing is allowed. If we do not want inputs to be shared, we create D_i without subtracting Σ. We denote the ordered set of all D_i by D.

Step 1: We assign integer $i := 1$;

Step 2: First, we assign boolean variable $m := false$; and then, we evaluate for D_i its intersections with all $D_j \in D$ such that $j \neq i, j \in |I|, |I| = |D|$. If any $D_i \cap D_j \neq \emptyset$ then we assign $D_j := D_j \cup D_i$; and $m := true$;

If $m = true$ after evaluating all intersection then we assign set $D_i := \emptyset$; to specify that D_i was split in other sets.

Step 3: We increment i by 1. If $i \leq |D|$ then we repeat Step 2, otherwise we delete from D all empty sets.

If we have deleted at least one empty set from D and $|D| > 1$, then we go to Step 1, otherwise to End.

End: If $|D| > 1$ then D contains the sets of variables that determine searched parallel decompositions. It always holds that $D_i \cap D_j = \emptyset$ for any $i, j \in I, |I| = |D|$ such that $i \neq j$, otherwise the algorithm could not terminate

We select any $D_i \in D$ and create parallel decomposition given by the equivalences of set of indexes D_i and $\mathrm{co}_P(\widehat{C}) - D_i$.

If $|D| > 2$ then the decomposition can be repeated until obtaining $|D|$ parallel automata.

4.2.2 Complexity and Correctness

The cardinality of $|D|$ is reduced at least by 1 at Step 3 when the algorithm continues. Thus, if we denote $n = |\mathrm{co}_P(\widehat{C})|$ then the algorithm always terminates at most after providing

$$\sum_{i=2}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} - 1$$

intersections in Step 2. Hence its complexity is $O(n^3)$.

If $|D| > 1$ when the algorithm has terminated, then $D_i \cap D_j = \emptyset$ for any $i, j \in I, |I| = |D|$. Because every D_i contains the unions of domains and codomains of all dependent t-assignments, the transfer sets

$$\widehat{C}_1 = D_i \mathbin{\widehat{\cap}} \widehat{C} \quad \text{and } \widehat{C}_2 = \widehat{C} - \widehat{C}_1$$

satisfy all conditions of Proposition 4.6.

Example 4.2

The example depicted in Figure 4.2 does not reflect a real technology exactly and it shows common imperfections of many examples. They are expected to be simple and comprehensible, but also very terse, otherwise they would not demonstrate anything interesting.

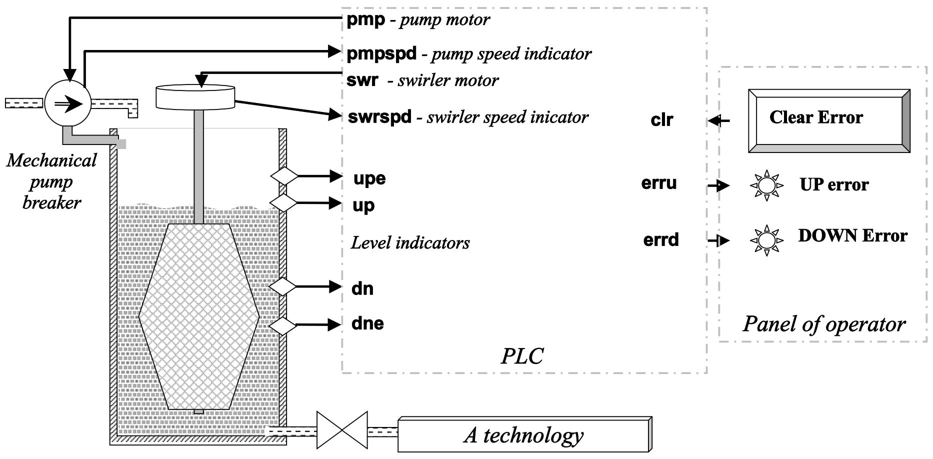


Figure 4.2: Example of Mix Tank

Suppose that we have a mix tank with swirler swr and pump pmp. They are equipped by centrifugal indicators $swrspd$ and $pmpspd$ that signal that related drive have reached an operating speed. The level of a liquid in the tank is watched by four indicators, of which dn and up serve as working sensors of normal minimal and maximal levels. These are duplicated by dne and upe indicators to increase safety. If liquid level has subsided to dne or climbed to upe, then something is out of order. Hence the alarm ($erru$ or $errd$) is announced on a remote operator panel until clr button is pressed. This approach assures drawing attention of operators who should inspect the reason of such malfunction.

To prolong lifetime of swirler screen, both pump and swirler should not run long at operating speeds concurrently. We verify neither $\mathrm{AG}\neg(swr \wedge pmp)$ nor $\mathrm{AG}\neg((swrspd \wedge pmp) \vee (swr \wedge pmpspd)$ CTL propositions, because concurrent motion of the swirler and pump is not a critical error. We should only check, whether the both devices do not operate too long at full power simultaneously. This vague formulation corresponds well to a non professional's statement of some verification problem.

The easiest solution leads to adding a check that will prevent too long concurrent motion of the pump and swirler. But we were not asked to rewrite a program. Our task consists in verifying the ladder diagram in Figure 4.4 (see page 116). Its PLC S7-210 statement list and corresponding APLC instructions are shown in Table 4.2 (see page 117). APLCTRANS gives result:

```
33 statements in 1 compositional block. 13 variables.
Composed in 0.14 s. Result=
{    @f[1], eu1[clr], clredge[clr.!eu1],
     pmp[dn.!swrspd.!up.!upe+dne.!swrspd.!up.!upe+pmp.!up.!upe],
     swr[!dn.!dne.!pmpspd+!dn.!dne.swr],
     erru[!clr.upe+!clr.erru+eu1.upe+eu1.erru],
     errd[!clr.dne+!clr.errd+eu1.dne+eu1.errd]     }
```

Extracting common parts in and-terms we obtain the transfer set:

$$\widehat{C} = \left\{ \begin{array}{ll} \widehat{f_{reg}} & [\![1]\!] \\ \widehat{eu_1} & [\![clr]\!] \\ \widehat{clredge} & [\![clr \wedge \neg eu_1]\!] \\ \widehat{pmp} & \left[\!\!\left[\neg up \wedge \neg upe \wedge \Big((\neg swrspd \wedge (dn \vee dne)) \vee pmp\Big)\right]\!\!\right] \\ \widehat{swr} & [\![\neg dn \wedge \neg dne \wedge (\neg pmpspd \vee swr)]\!] \\ \widehat{erru} & [\![\neg(clr \wedge \neg eu1) \wedge (upe \vee erru)]\!] \\ \widehat{errd} & [\![\neg(clr \wedge \neg eu1) \wedge (dne \vee errd)]\!] \end{array} \right\}$$

We calculate its domain and PLC codomain:

$$\begin{array}{rcl} \mathrm{dom}(\widehat{C}) & = & \left\{ \begin{array}{l} clr, eu_1, up, upe, dn, dne, \\ swr, swrspd, pmp, pmpspd, erru, errd \end{array} \right\} \\ \mathrm{co}_P(\widehat{C}) & = & \{eu_1, clredge, pmp, swr, erru, errd\} \\ W & = & \mathrm{co}_P(\widehat{C}) \cap \mathrm{dom}(\widehat{C}) = \{eu_1, swr, pmp, erru, errd\} \quad (4.12) \end{array}$$

which yield 32 states $Q = \alpha(W)$ of generated automaton (Proposition 4.1 on page 101).

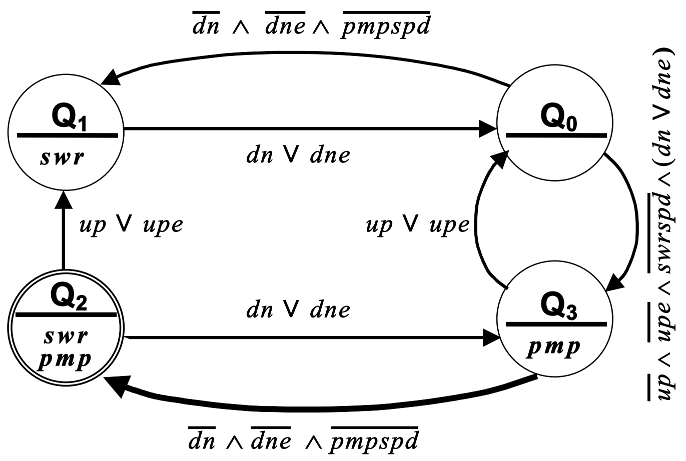
We try its parallel decomposition. We create set D (Equation 4.11 on page 111) and list its members in the following table together with D contents at the beginning of Step 3.

Initilize		Step 3: $i = 1$	...	$i = 5$	$i = 6$
$D_1 =$	eu_1	$\emptyset$		$\emptyset$	$\emptyset$
$D_2 =$	$clredge,\ eu_1$	$clredge,\ eu_1$		$\emptyset$	$\emptyset$
$D_3 =$	pmp	pmp		pmp	pmp
$D_4 =$	swr	swr		swr	swr
$D_5 =$	$erru,\ eu_1$	$erru,\ eu_1$		$erru,\ eu_1,\ clredge$	$\emptyset$
$D_6 =$	$errd,\ eu_1$	$errd,\ eu_1$		$errd,\ eu_1,\ clredge$	$erru,\ errd,\ eu_1,\ clredge$

The result offers 3 possible decomposition to two automata and 1 to three automata, but we verify concurrent behavior of the pump and swirler, therefore we may chose only one decomposition defined by $D_3 \cup D_4$ and D_6. [2]

Hence we will verify the automaton given by

$$\widehat{C}_1 = \left\{ \begin{array}{ll} \widehat{pmp} & \left[\!\!\left[\neg up \wedge \neg upe \wedge \Big((\neg swrspd \wedge (dn \vee dne)) \vee pmp\Big)\right]\!\!\right] \\ \widehat{swr} & [\![\neg dn \wedge \neg dne \wedge (\neg pmpspd \vee swr)]\!] \end{array} \right\}$$



$\overline{dn} \wedge \overline{dne} \wedge \overline{pmpspd}$
Q_1
swr
$dn \vee dne$
Q_0
$up \vee upe$
$up \vee upe$
$\overline{up} \wedge \overline{upe} \wedge \overline{swrspd} \wedge (dn \vee dne)$
Q_2
swr
pmp
$dn \vee dne$
Q_3
pmp
$\overline{dn} \wedge \overline{dne} \wedge \overline{pmpspd}$

Figure 4.3: Decomposed Automaton of Mix Tank

The automaton could be inspected by some model checker, but we depict its diagram for illustration in Figure 4.3, in which logical negations are denoted by overlines for better readability. The program violates the requirement by the transition which is emphasized by bold line. This situation occurs under following conditions:

1. the level of liquid is above dn and dne indicators,
2. the pump is operating,
3. but pump speed fall down for a moment and its speed indicator gives short 0 pulse, for example due to sediments in liquid.

In this case the pump will run with the swirler concurrently until the level of liquid rises to upper indicators.

The probability of occurring of this situation above depends on running characteristics of the pump, swirler and speed indicators. But such malfunction could appear very rare under fatal conditions, which would means that it need not be detected by program tests.

[2]Notice that D_i specifies only variables which t-assignments are included in a decomposition, for example D_6 leads to an automaton with 8 states (see Equation 4.12).

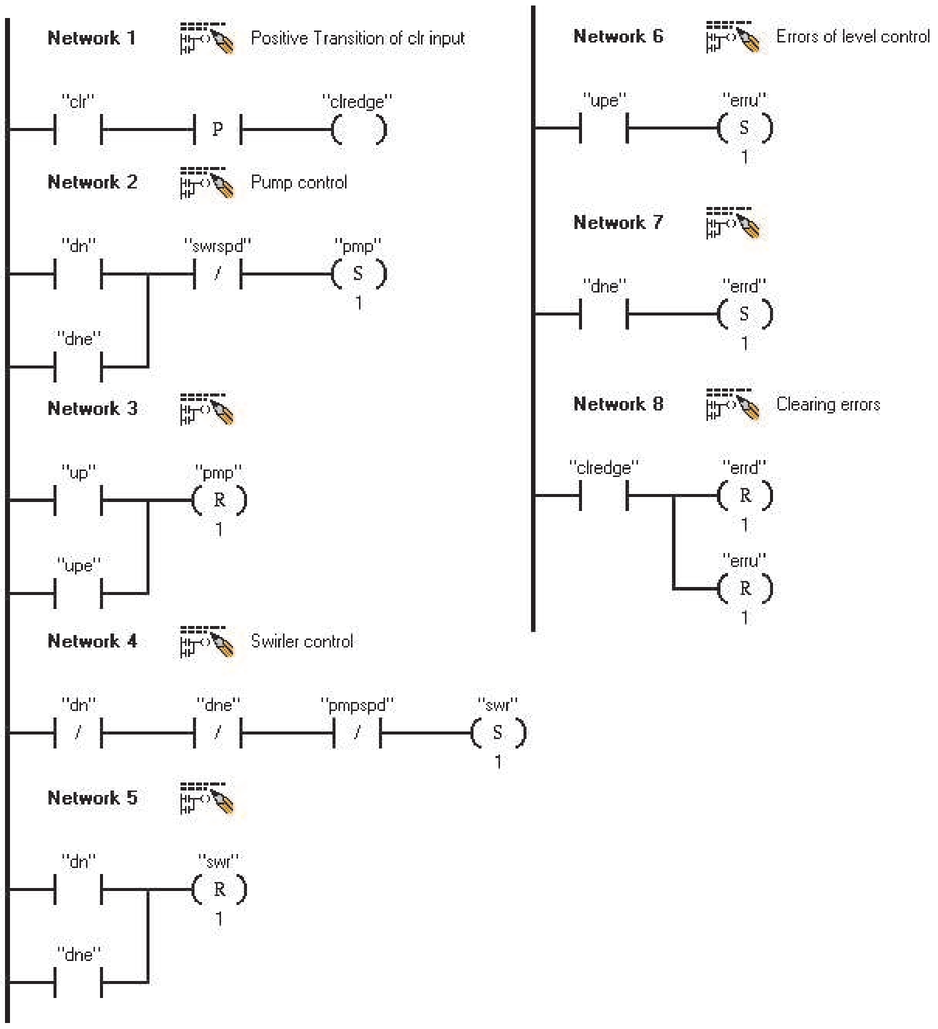


Figure 4.4: S7-210 Ladder Diagram of Mix Tank

APLC program	**S7-210 STL**	**Comment**
Init;	NETWORK 1	*Positive Transition of clr input*
Load clr;	LD "clr"	
REdge eu1;	EU	*EU uses hidden variable - it was added*
Store clredge;	= "clredge"	
Init;	NETWORK 2	*Pump control*
Load dn;	LD "dn"	
Or dne;	O "dne"	
And !swrspd;	AN "swrspd"	
Set pmp;	S "pmp", 1	
Init;	NETWORK 3	
Load up;	LD "up"	
Or upe;	O "upe"	
Res pmp;	R "pmp", 1	
Init;	NETWORK 4	*Swirler control*
Load !dn;	LDN "dn"	
And !dne;	AN "dne"	
And !pmpspd;	AN "pmpspd"	
Set swr;	S "swr", 1	
Init;	NETWORK 5	
Load dn;	LD "dn"	
Or dne;	O "dne"	
Res swr;	R "swr", 1	
Init;	NETWORK 6	*Errors of level control*
Load upe;	LD "upe"	
Set erru;	S "erru", 1	
Init;	NETWORK 7	
Load dne;	LD "dne"	
Set errd;	R "errd", 1	
Init;	NETWORK 8	*Clearing errors*
Load clredge;	LD "clrmem"	
Res errd;	R "errd", 1	
Res erru;	R "erru", 1	
End		

Table 4.2: APLC Mix Tank Program

4.3 Detecting Races

The designation "races" was presented by Aiken, Fähndrich, and Su in [AFS98]: "If under fixed inputs and timer and counter states, an output x changes from scan to scan, then there is a *relay race on* x". The authors have developed the algorithm capable of finding races, but not proving their absence:

> *The absence of races cannot be proven by our analysis due to approximations and due to the finite subspace of input assignments that are sampled.* (quoted from [AFS98, page 10])

In contrast, if PLC program is convertible to APLC language, then APLCTRANS proves the absence of races. Before presenting the definition, we give the explanation of the races.

The simplest race is an assignment: $x = \neg x$ because PLCs operate in endless loops so the assignment is in fact executed as:

RaceA:	LOAD	$\neg x$;		**repeat**
	STORE	x;	$\Rightarrow$	x := NOT x;
	END;			**until** *false*;

The transfer set of *RaceA* program is $\widehat{C}_A = \hat{x} \llbracket \neg x \rrbracket$.

In the terms of the automaton theory, a race represents instable state, from which an automaton moves to another state in next time step, though inputs have not changed. However, not all races are errors, they are sometimes created intentionally. For example, an edge detection will result in a race. We apply one for improving *RaceA* program.

To avoid jump, we replace $\neg$ by non equivalence $\neg x = x \not\equiv 1$ and we utilize the fact that *yedge* equals to 1 only for such scan, in which y went from 0 to 1.

RaceB:	LOAD	y;		**repeat**
	REDGE	*ymem*;		yedge := (y AND (NOT ymem));
	STORE	*yedge*;	$\Rightarrow$	ymem := y;
	LOAD	$x \not\equiv$ *yedge*;		x := x XOR yedge;
	STORE	x;		**until** *false*;
	END;			

APLCTRAN result is:

```
Composed in 0.01 seconds. Decoding result...
Result={ ymem[y], yedge[y.!ymem], x[y.!ymem.!x+!y.x+ymem.x] }
```

Now, the execution of $x = \neg x$ is synchronized by a rising edge of another input y and it is performed only once when y goes from 0 to 1. After rewriting formula for x the transfer set of *RaceB* program is:

$$\widehat{C}_B = \left\{ \begin{array}{l} \widehat{ymem}\,[\![y]\!]\,, \widehat{yedge}\,[\![y \wedge \neg ymem]\!]\,, \\ \hat{x}\,[\![(y \wedge \neg ymem \wedge \neg x) \vee \neg((y \wedge \neg(ymem) \vee \neg x)]\!] \end{array} \right\}$$

RaceA program without the edge detection has endless race in x. The improved program *RaceB* contains the race in *yedge*, but this race will disappear after next program scan.

All edge detections represent first mathematical differences of inputs and their usage is necessary, because many operations require synchronization to assure their single executions. Hence, the most of PLC programs have races with the length 1 i.e., the races that disappear after 1 next scan.

Longer races usually indicate an inefficient coding. For example, we can insert in front of *RaceB* program four statements:

```
RaceC   LOAD    q1;
        STORE   q2;     ⇒        q2:=q1;
        LOAD    x;               q1:=x;
        STORE   q1;
RaceB:  LOAD    y;    ...
```

We have obtained *RaceC* program with transfer set:

$$\widehat{C}_C = \widehat{C}_B \cup \{q_1\,[\![x]\!]\,, q_2\,[\![q_1]\!]\}$$

RaceC program contains several races. If positive edge of y is detected, then *yedge* goes to 1 for one scan and x is negated. But changed x value will propagate to q_1 in the second scan and to q_2 in the third scan. Hence, the lengths of races are 1 for *yedge* 1, 2 for q_1, and 3 for q_2.

However, *RaceC* program represents a nonsensical code. The output q_2 reacts to a change of y with a delay of three scans that could have random length because PLC scans are usually irregular. Similar operations are also clumsy — if an output must be delayed then timers offer more reliable solutions, which will be moreover clear at first glance when someone looks at the source code.

If we execute these four instructions after *RaceB* and swap their order, then new program will evaluate the same result, but without any random delay.

```
RaceD:  JS      RaceB;
        LOAD    x;
        STORE   q1;
        LOAD    q1;
        STORE   q2;
        END;
RaceB:  LOAD    y;
                ...
```

RaceD has the transfer set

$$\widehat{C}_D = \widehat{C}_B \cup \left\{ \begin{array}{l} \hat{q}_1 \,[\![(y \wedge \neg ymem \wedge \neg x) \vee \neg((y \wedge \neg(ymem) \vee \neg x))]\!] \,, \\ \hat{q}_2 \,[\![(y \wedge \neg ymem \wedge \neg x) \vee \neg((y \wedge \neg(ymem) \vee \neg x))]\!] \end{array} \right\}$$

where the both expressions equal to the expression of $\hat{x}\,[\![bexp_x]\!] \in \widehat{C}_B$.

It follows from previous considerations that 1 scan races are programmed intentionally in most cases, but longer races will signal with higher probability that "something is rotten in the states of PLC". They are either program errors or signs of improper coding.

Definition 4.8 *Let $AL \in AProgram$ be any APLC program defined on $S = \{\Sigma, V, \Omega\}$ storage and given its transfer set $\widehat{C}$ created by APLCTRANS. If k is the least number satisfying:*

$$\widehat{\prod}_{i=1}^{k+1} \widehat{C}_W \mathrel{\widehat{=}} \widehat{\prod}_{i=1}^{k+2} \widehat{C}_W \quad \textit{where } \widehat{C}_W = \widehat{C} \mathbin{\widehat{\cap}} (V \cup \Omega) \tag{4.13}$$

then we say that AL has race of k-th order.

We have based the definition on the transfer set, from which have been removed all t-assignments of inputs Σ. The necessity of this intersection follows from the considerations discussed on page 98. Some operations can change the contents of input variables, because they are stored in input image, but the *next scan always rewrites inputs* by their new values. If no input has changed, then all inputs are reset to canonical t-assignments. Removing input t-assignments gives the same result due to $\uparrow$ operator properties (see pages 61 and 64).

Definition 4.8 also gives the algorithm for testing races. The results can lead to large functions. Checking equivalences allows Stålmarck's patented proof procedure for propositional logic [SS98] that has worst case complexity exponentially depended on depth of formula. In many cases, we do not need check all equivalences.

Proposition 4.7 *Let $\widehat{C}$ and $\widehat{C}_r$ be transfer sets defined on $S = \{\Sigma, V, \Omega\}$ and $\hat{x} \in \widehat{C}$ any given t-assignment. If $\hat{x}$ satisfies either*

$$\mathrm{dom}(\hat{x}) \subset \Sigma \tag{4.14}$$

or all three following conditions:

$$\begin{aligned} \hat{x} \circ \hat{x}\,[\![1]\!] &\mathrel{\widehat{=}} \neg\,(\hat{x} \circ \hat{x}\,[\![0]\!]) \\ \mathrm{dom}(\hat{x}) - \Sigma &= \mathrm{co}(\hat{x}) \\ \hat{x} &\mathrel{\widehat{=}} \left(\mathrm{co}(\hat{x}) \mathbin{\widehat{\cap}} \widehat{C}_r\right) \end{aligned} \tag{4.15}$$

then $\hat{x} \in (\widehat{C} \circledcirc \widehat{C}_r)$.

Proof: Equation 4.14 specifies that $\hat{x}$ should only depend on inputs. They are canonical t-assignments according to Equation 4.13 i.e., members of $\widehat{\mathcal{E}}^S$, which is an identity element of t-assignments monoid (Proposition 3.9). Hence, $\hat{x}$ value will not change.

The second and third conditions of Equation group 4.15 require that $\hat{x}$ depends only on itself and it is $\widehat{=}$ equal in the both transfer sets $\widehat{C}$ and $\widehat{C}_r$. Hence, $\circledcirc$ composition will substitute $\hat{x}$ into itself as $\hat{x} \circ \hat{x}$.

To prove the first condition of Equation group 4.15, we rewrite $\hat{x}$ with the aid of Shannon's expansion:

$$\hat{x}\,[\![bexp_x]\!] \quad = \quad \hat{x}\,[\![(x \wedge v_1) \vee (\neg x \wedge v_0)]\!] \tag{4.16}$$

where $v_1 = \hat{x} \circ \hat{x}^1\,[\![1]\!]$ and $v_0 = \hat{x} \circ \hat{x}^0\,[\![0]\!]$ evidently.

Now we substitute $bexp_x$ into itself. Let us write $bexp_y$ for the value of $\hat{y} = \hat{x} \circ \hat{x}$. It yields

$$\begin{aligned} bexp_y &= ((x \wedge v_1) \vee (\neg x \wedge v_0)) \wedge v_1) \vee (\neg\,((x \wedge v_1) \vee (\neg x \wedge v_0)) \wedge v_0) \\ &= (x \wedge v_1) \vee (x \wedge v_0) \vee (v_1 \wedge v_0) \qquad (4.17) \end{aligned}$$

We want $bexp_y \equiv bexp_x$, which gives the condition

$$\begin{aligned} 1 &= (bexp_y \equiv bexp_x) = (bexp_y \wedge bexp_x) \vee (\neg bexp_y \wedge \neg bexp_x) \\ &= \Big(\, ((x \wedge v_1) \vee (x \wedge v_0) \vee (v_1 \wedge v_0)) \wedge ((x \wedge v_1) \vee (\neg x \wedge v_0)) \,\Big) \\ &\quad \vee \Big(\, \neg((x \wedge v_1) \vee (x \wedge v_0) \vee (v_1 \wedge v_0)) \wedge \neg((x \wedge v_1) \vee (\neg x \wedge v_0)) \,\Big) \\ 1 &= v_1 \wedge \neg v_2 \qquad (4.18) \end{aligned}$$

Applying Equation 4.18 together with Equation 4.16 yields Equation 4.15. If $\hat{x}$ satisfies this condition together with the other ones of the group, then it will not change its value by composition. Therefore, it will hold that $\hat{x} \in (\widehat{C} \circledcirc \widehat{C}_r)$. □

Note: Equation 4.18 can be also derived by APLCTRANS. We verify the proof by the composition of APLC program:

```
Proof: LOAD  (x ∧ v1) ∨ (x ∧ v0);     STORE  x;
       LOAD  (x ∧ v1) ∨ (x ∧ v0);     STORE  y;
       LOAD  (x ∧ y) ∨ (¬x ∧ ¬y);     STORE  r;
       END;
```

```
Composed in 0.00 seconds. Decoding result...
Result={ r[v1+!v0], x[x.v1+!x.v0], y[x.v1+v1.v0+x.v0] }
```

APLCTRANS tool can convert results directly to $\LaTeX\,2_\varepsilon$:

$\widehat{C_p} = \{\hat{r}\,[\![v_1 \vee \neg v_0]\!]\,, \hat{x}\,[\![(x \wedge v_1) \vee (\neg x \wedge v_0)]\!]\,, \hat{y}\,[\![(x \wedge v_1) \vee (v_1 \wedge v_0) \vee x(\wedge v_0)]\!]\}$

Proposition 4.7 allows to determine that some t-assignments cannot have a race, but it does not say, which t-assignment have a race. Moreover, it requires satisfying many conditions, hence it excludes only small part of t-assignments. The other must be checked as equivalences.

Example 4.3

We analyze races in $\widehat{C}_B$. Suppose that storage

$$S = \{\Sigma = \{y\}, V = \{ymem, yedge\}, \Omega = \{x\}\}$$

then $ymem$ satisfies Equation 4.14. It has no race certainly. The remaining function must be checked by equivalences. Because $\widehat{C}_B$ do not contain any t-assignments of Σ variables, we modify $RaceB$ to evaluate Equation 4.13:

```
        JS    RaceB;
        JS    RaceB;
        END
RaceB:  ...
```

APLCTRANS result is:

```
Composed in 0.01 seconds. Decoding result...
Result={ ymem[y], yedge[0], x[!y.x+y.!ymem.!x+ymem.x] }
```

We see that $ymem$ has no race as expected, but $yedge$ and x produce the race. Now $yedge$ satisfies Equation 4.14, because its domain is empty set. Inserting another JS $RaceB$ instruction yields the same result as above. Hence, program have only normal races with length 1 generated by edge detection.

We also analyze the program in Example 4.2 (see page 112), whose transfer set is on page 114. We can apply Equation 4.14 to eu_1. The pump and swirler satisfy the second and third condition of Equation group 4.15. Using the first condition for the pump we obtain:

$$\begin{aligned}
pmp &= \neg up \wedge \neg upe \wedge \Big((\neg swrspd \wedge (dn \vee dne)) \vee pmp \Big) \\
v_0 &= \neg up \wedge \neg upe \wedge \neg swrspd \wedge (dn \vee dne) \\
v_1 &= \neg up \wedge \neg upe \\
v_1 \vee \neg v_0 &= v_1 \vee \neg(v_1 \wedge \neg swrspd \wedge (dn \vee dne)) \\
&= v_1 \vee \neg v_1 \vee \neg(\neg swrspd \wedge (dn \vee dne)) \\
&= 1
\end{aligned}$$

Applying similar procedure to the swirler also proves that swr is race-free.

Hence, eu_1, pmp, and swr have no race. The other variables require testing by Equation 4.13

```
Composed in 0.03 seconds. Decoding result...
Result={ eu1[clr], clredge[0],
pmp[dn.!swrspd.!up.!upe+dne.!swrspd.!up.!upe+pmp.!up.!upe],
swr[!dn.!dne.!pmpspd+!dn.!dne.swr],
erru[!clr.erru+upe+eu1.erru],
errd[!clr.errd+dne+eu1.errd] }
```

The following compositions give the same transfer set. Hence, the program has races of the first order.

4.4 Abstract PLC and Timed Automata

We have mentioned on page 28 that timers substitute for an incomplete knowledge of the PLC environment, for example some operations can be checked by measuring their time. PLC programs utilize timers frequently and models, which do not consider time behavior, have either limited application or require skilful preparations of control program to allow its partial verification.

APLC machine does not contain any elements for expressing time characteristics. In reality, they have not been added, because they are not needed for converting PLC programs to timed automata. In this section, we explain how APLCTRANS could provide such conversions and we will demonstrated them by a simple example.

Timed finite automata were proposed by Alur and Dill in [AD94]: "to model the behavior of real timed system over time". Their timed automata are based on ω-automata, mainly on Büchi and Muller types. They may be derived from automata of Mealy's family by adding the set of starting states and the set of acceptance conditions. These automata recognize infinite ω-languages (see for instance [Pel00]).

Timed automata are intensively studied (selected references [Alu00], [DY96], [SV96], and [LLW95]). Their verification methods can be found for instance in [MLAH01], [Yov98], [AJ98], [BMPY97], and [SVD97].

The definition of timed automaton in [AD94] is based on several concepts, which we briefly outline.

The fundamental element is the set of clocks T, which represents the set of real values that are increasing strictly monotonically with real-time. All these clocks run independently, because their increasing is not controlled by the timed automaton. It mays reset a clock $\tau_i \in T$ or block its progress, but timed automaton does not up clocks.

The set $\Phi(T)$ of clock constraints $\varphi \in \Phi(T)$ is defined inductively by [3]

$$\varphi \overset{df}{=} \tau \leq c | c \leq \tau | \neg\varphi | \varphi_1 \wedge \varphi_2 \tag{4.19}$$

where $\tau \in T$ and c stands for some nonnegative (possible different) rational constants. $\Phi(T)$ expressions are analogous to the ones generated by *Bexp* grammar.

If we replace boolean variables in *Bexp* grammar by time constraints, we would obtain similar language, because any logical function can be expressed with the aid of $\neg$ and $\wedge$ operators.

The clock constraints are employed in timed transition table [AD94, page 9], which could be considered as some replacement of δ transition function that expresses behavior of Mealy's automata.

[3]In Alur and Dill's paper is used $\Phi(C)$ instead of $\Phi(T)$ and δ instead of φ. We have changed notation to distinguish them from symbols that we gave to other elements.

Definition 4.9 *A timed transition table $\mathcal{A}$ is a tuple $\langle \Sigma, S, S_0, T, E \rangle$, where*

- *Σ is a finite alphabet,*
- *S is a finite set of states,*
- *$S_0 \subseteq S$ is a set of start states,*
- *T is a finite set of clocks, and*
- *$E \subseteq S \times S \times 2^C \times \Phi(T)$ gives the set of transitions. An edge $\langle s, s', a, \lambda, \varphi \rangle$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq T$ gives the clocks to be reset with this transition, and φ is a clock constraint over a set of clock variables T.*

Other authors extended the definition by another elements as enabling conditions for clocks [LLW95, page 2], which enable or disable the progress of individual clocks. Timed automaton definition follows from the transition table [AD94, page 10]:

Definition 4.10 *A* timed Büchi automaton **U** *is a tuple $\langle \Sigma, S, S_0, T, E, F \rangle$, where $\langle \Sigma, S, S_0, T, E \rangle$ is a transition table, and $F \subseteq S$ is a set of* accepting *states.*

If we replace all clock constraints by some inputs, then the timed automaton defined above will change to an automaton of Mealy's family with external clock modules, whose progress is controlled by auxiliary outputs as depicted in Figure 4.5. APLCTRANS is able to compose it and convert the result into a timed automaton.

The interesting question is certainly about the relation between PLC software timers and their timed models. Unfortunately, *the theory ends here and engineering know-how begins.* Timed model of particular timer depends not only on its type and manufactured PLC type, but also on methods how PLC program accesses variables related to the timer. Many timers allow simple conversions, but more complicated models are sometimes required.

A universal model would probably exist, to which PLC timers could be transformed, but such overall model could generate many states needlessly, even if they are not necessary.

Because many different PLC timers exist, we have selected two most frequent types, for which we outline the conversion of PLC to timed automaton. *Timer on delay* delays rising edge of its input and *timer off delay* prolongs falling edge. The both timers generate binary outputs DN and TT as shown in Figure 4.6.

Other PLC timer types are mostly derived from on and off delay timers, with the exception of retentive timer. It is required for special processes, whose operation could be possibly divided into several time intervals by miscellaneous interruptions, but the total time of their run may not expire

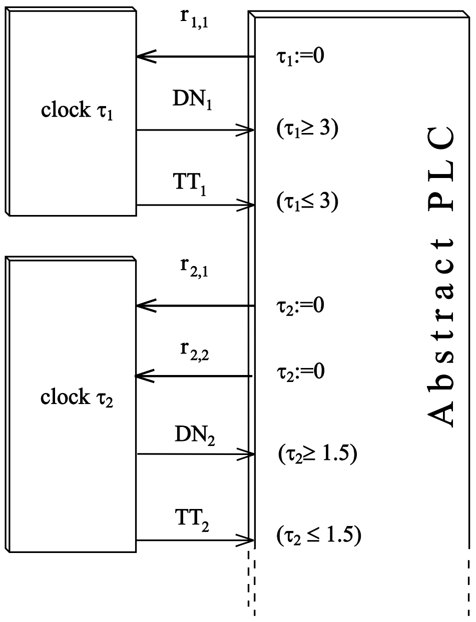


Figure 4.5: Untimed Abstract PLC with External Timers

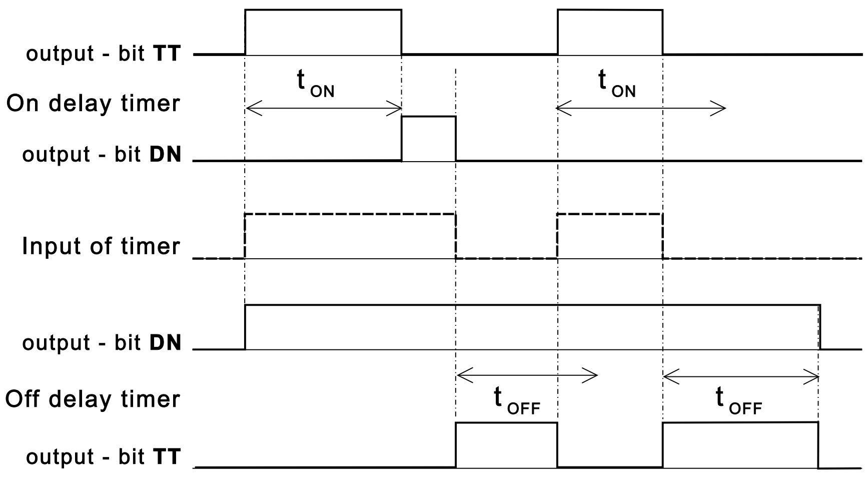


Figure 4.6: PLC Timers On Delay and Off Delay

a preset time. Retentive timer leads to timed automata equipped with enabling and disabling of individual clocks. However, its conversion would be similar.

4.4.1 Outline of APLC Conversion to Timed Automaton

Modifications Performed Before APLCTRANS

Each timer in a PLC program stands for one clock, thus the set of all clocks is $T = \{\tau_1, \ldots, \tau_{n_t}\}$ where n_t is the total number of timers in a program. Any timer is primary controlled by its timer instruction, but some PLCs may allow resetting timers by special instructions. We will call these instructions (i.e., timer and resetting instructions) *control accesses to timer τ_i*. Resetting clocks on the edges of a timed automaton corresponds to APLC rising or falling edge detections (REdge or FEdge instructions).

Inserting these detection into APLC program before its composing would hide information, which boolean expression belongs to specific timer. Therefore, we create new boolean variable for each control access to the timer τ_i. It gives the set of control accesses:

$$\gamma^r(\tau_i) \overset{df}{=} \{r_{i,j} \in \mathcal{B} \mid j \in I, |I| = m_i\}$$

where m_i represents the number of control accesses to τ_i. We reserve $r_{i,1}$ for the timer instruction and assign $r_{i,j}, j > 1$ to reset instructions. We remember c_i time constant of τ timer and replace all control accesses to timer τ_i by m_i instructions Store $r_{i,j}$.

Let us write Γ for the set of all $r_{i,j}$:

$$\Gamma = \bigcup_{i=1}^{n_t} \gamma^r(\tau_i) \text{ where } \tau_i \in T$$

Notice, that each $r_{i,j}$ is used only once in the modified program, because every substituted Store has different operand $r_{i,j}$.

APLCTRANS Run

We have replaced all timers by operations with boolean variables, which results in APLC program. Hence APLCTRANS can compose it to transfer set $\widehat{C}$. Because $r_{i,j}$ were used only once in Store instruction, the value of t-assignment $\hat{r}_{i,1}$ contains logical condition of timer τ_i input and the values of $\hat{r}_{i,j}$, $j > 1$ belongs to reset instructions.

$\widehat{C}$ is possible to decompose to parallel automata by the equivalences of set of indexes, with the aid of the algorithm presented in Section 4.2, if this is needed. We must only select such union of $D_i \in D$, which includes all variables belonging to one timer (see page 112).

Modification of APLCTRANS Result

To simplify modification, we replace all $\hat{r}_{i,j}$ in $\widehat{C}$ by canonical t-assignments $\hat{r}'_{i,j}$ and we denote the set of all $\hat{r}'_{i,j}$ by Γ' and new transfer set by $\widehat{C}'$.

We create the domains and codomains of $\widehat{C}'$ according to Definitions 4.2 and 4.3 (see page 97) and we add all Γ' variables to the domain and codomain to make them state variables. We must do that, because the domain and codomain definition do not include canonical t-assignments.

This yields the product of APLC automaton with simple automata shown in Figure 4.7 allowing to assign reset clock operations to the edges.

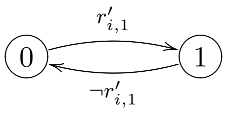


Figure 4.7: Automata Added for Timers

Using these modified domains and codomains of $\widehat{C}'$ we create the automaton of Mealy's family according to Definition 4.1. We list all its edges and add time constraints φ defined by Equation 4.19.

First, we assign reset operations by replacing all $r'_{i,j}$ according to the type of original instruction. The substitutions are shown in the table below, where $bexp_{i,1}$ stands for the value (i.e. expression) of t-assignment $\hat{r}_{i,j}$ $[\![bexp_{i,1}]\!]$:

reset instruction	$r'_{i,j}$	$bexp_{i,1} \wedge (\tau_i := 0)$
$j > 1$	$\neg r'_{i,j}$	$\neg(bexp_{i,1})$
on delay timer	$r'_{i,1}$	$bexp_{i,1} \wedge (\tau_i := 0)$
	$\neg r'_{i,j}$	$\neg(bexp_{i,1})$
off delay timer	$r'_{i,1}$	$bexp_{i,1}$
	$\neg r'_{i,j}$	$\neg(bexp_{i,1}) \wedge (\tau_i := 0)$

The automaton in Figure 4.7 is now modified to one of two automata in Figure 4.8

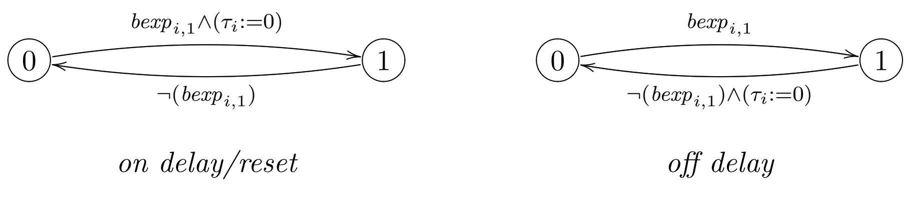

$bexp_{i,1} \wedge (\tau_i := 0)$
0
1
$\neg(bexp_{i,1})$
on delay/reset
$bexp_{i,1}$
0
1
$\neg(bexp_{i,1}) \wedge (\tau_i := 0)$
off delay

Figure 4.8: Reset, On and Off Delay Timer Automata

Finally, we modify accesses to timer bits TT an DN that are tested in the program. We will supposed that timer bits TT and DN are read only and program instruction do not change them directly, for example by Store, Set, or Res instructions. [4]

We replace DN and TT timer bit by formulas $[\neg]_1 \mathit{bexp}_{i,1} \odot [\neg]_2 \varphi$ where

- $\mathit{bexp}_{i,1}$ is the value of $\hat{r}_{i,1}$,
- $[\neg]_k$ denotes optional boolean negation,
- $\odot$ is boolean operation, either $\wedge$ or $\vee$, and
- φ is timing constraint, in which is utilized timing constant c_i.

The formulas for timer bits are:

τ_i	on delay timer	DN bit	$(\mathit{bexp}_{i,1})$	$\wedge$	$(\tau_i \geq c_i)$
		TT bit	$(\mathit{bexp}_{i,1})$	$\wedge$	$(\tau_i \leq c_i)$
	off delay timer	DN bit	$(\mathit{bexp}_{i,1})$	$\vee$	$\neg(\tau_i \geq c_i)$
		TT bit	$\neg(\mathit{bexp}_{i,1})$	$\wedge$	$(\tau_i \leq c_i)$

Minimization of Timed Automaton

We inspect the edges that come out from the same state. We search for two edges such one contains resetting of a clock $(\mathit{bexp}_1) \wedge \tau_i := 0$, where bexp_1 is a boolean function, and the second edge contains a time constraint $(\mathit{bexp}_1) \wedge (\mathit{bexp}_2) \wedge (\tau_i \geq c_i)$, where bexp_2 is another boolean function (possibly equal to 1).

If we locate such two edges, we delete the second one, because it is redundant. When bexp_1 reset a clock, then $(\tau_i \geq c_i)$ will never hold in PLC timer.

Discussion

The formulas for representing timer operations will vary with PLC type and the expertise is always necessary to give precise specifications for a manufactured PLC. Therefore, we have presented this conversion only as an outline, not an algorithm.

Further research is required to establish exacter algorithmic method, which will applicable to wider range of PLC types without many modifications.

[4]Direct changing timer bits is somewhere possible on selected PLC types for some timers, but mostly not recommended, because such non standard operations complicate troubleshooting by more difficult orientation in a PLC program. If they were used after all, then such timers would have required more complex models, in which each timer τ_i will latch and unlatch an ordinary boolean variable instead of outputting read only DN bit.

Example 4.4

We will demonstrate converting timers to a timed automaton by the following simple example, in which SLC 5/02 TON timer detects a bad contactor operation. The ladder diagram of the program is depicted in Figure 4.9 together with its exported PLC source code and APLC instructions.

Contactor m has the secondary contact that confirms its closing. If m was set to 1, then m_2 should be closed at most after 3 second delay, otherwise the error must be announced by setting $error$ bit. This bit is periodically read by the main part of the program, which also select an proper action according to the state of a controlled technology.

Similar error detections simplify troubleshooting and a PLC program usually contains many timed checks. After their verification, it is sometimes possible to remove them from PLC model or to represent all these part by one block, which reduces the states significantly.

We proceed by creating timed automaton. The preparation phase means replacing one on delay timer instruction by $\text{S}\small{\text{TORE}}$ $r_{1,1}$ and remembering its time constant $c_1 = 3$ [s]. The set of the clocks has one element, $T = \{\tau_1\}$. Another necessary input information is the storage, on which is the program defined, to distinguish inputs from variables stored in memory.

$$S = \{\Sigma = \{s_1, m_2, dn_1\}, V = \{error, m, r_{1,1}\}, \Omega = \emptyset\}$$

Notice, that *we have added two system variables to* Σ — first scan signal s_1 (see page 21) and DN bit of timers. The both variables are stored in PLC memory, but they are read only data for its programs, i.e. the inputs of our automaton.

APLCTRANS composes the program to:

```
Result={ @f[1], @r11[m], error[@dn1.!m2+!s1.error] }
```

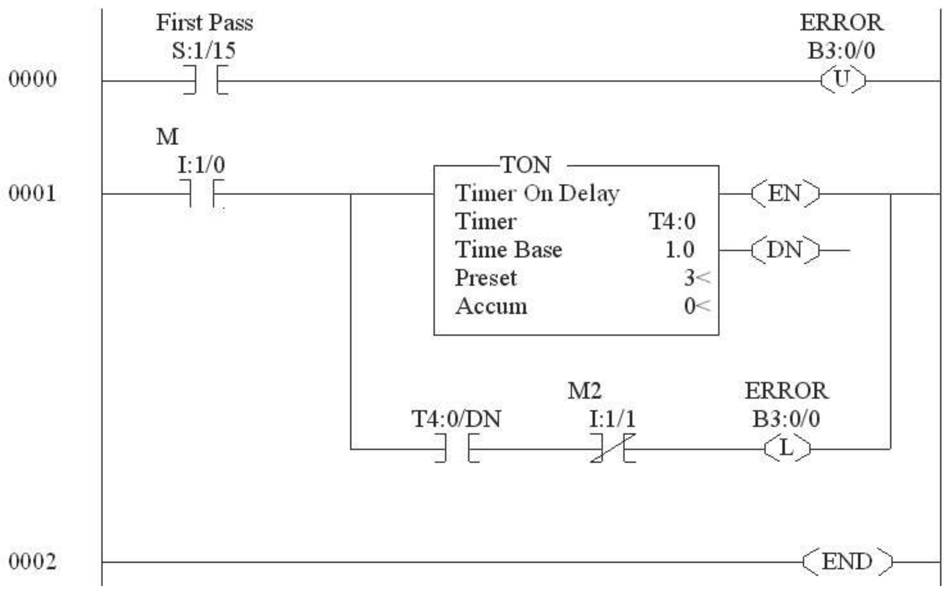
The @ prefixes were used for automatically created variables to distinguish them from other program labels. The result corresponds to the transfer set:

$$\widehat{C} = \{\hat{f}_{reg}\,[\![1]\!]\,, r_{1,1}\,[\![m]\!]\,, \widehat{error}\,[\![(dn_1 \wedge \neg m_2) \vee (\neg s_1 \wedge error)]\!]\}$$

Before we build its domain and PLC codomain, we modify $\widehat{C}$ to $\widehat{C}'$ by replacing $\hat{r}_{1,1}\,[\![m]\!]$ by canonical t-assignment $\hat{r}'_{1,1}\,[\![r'_{1,1}]\!]$ not to lose information about timer variables.

$$\begin{aligned}
\widehat{C}' &= \{\hat{f}_{reg}\,[\![1]\!]\,, \hat{r}'_{1,1}\,[\![r'_{1,1}]\!]\,, \widehat{error}\,[\![(dn_1 \wedge \neg m_2) \vee (\neg s_1 \wedge error)]\!]\} \\
\mathrm{dom}(\widehat{C}') &= \{r'_{1,1}, dn_1, m_2, s_1, error\} \\
\mathrm{co}_P(\widehat{C}') &= \{error\}
\end{aligned}$$

We see that $r_{1,1}$ would not be normally a state variable, because canonical t-assignments are automatically removed by $\downarrow$ operator. So we must add $r'_{1,1}$



APLC code	**SLC 5/02 PLC**	*Comment*
ladder2:		
INIT;	SOR	
AND $s1$;	XIC S:1/15	*First scan after switching PLC on*
RET *error*;	OTU B3:0/0	*Reset error*
INIT;	EOR	
INIT;	SOR	
AND m;	XIC I:1.0/0	*Is contactor m closed?*
PUSH; EPUSH 1;	BST	*Branch start*
Store $\mathbf{r_{1,1}}$;	TON T4:0 1.0 3 0	*on delay timer $c_1 = 3$ seconds*
TOR; DUP; ESWP	NXB	*next branch*
And $\mathbf{dn_1}$;	XIC T4:0/DN	*DN bit — 3 seconds has passed*
AND $!m_2$;	XIO I:1.0/1	*2nd contact has not closed?*
SET *error*;	OTL B3:0/0	*Set error*
TOR; DROP;	BND	*branch end*
INIT;	EOR	
END;	*end of file*	

Figure 4.9: Checking Contactor M with Secondary Contact M2

to domain and PLC codomain to obtain automaton containing the edges for resetting clocks.

$$\begin{aligned} \mathrm{dom}(\widehat{C}') &= \{\mathit{dn_1}, m_2, s_1, \mathit{error}\} \cup \{r'_{1,1}\} \\ \mathrm{co}_P(\widehat{C}') &= \{\mathit{error}, r'_{1,1}\} \end{aligned} \tag{4.20}$$

Using them we create the automaton of Mealy's family. The set of the state variable is intersection of domain and PLC codomain

$$W = \mathrm{dom}(\widehat{C}') \cap \mathrm{co}_P(\widehat{C}') = \{r'_{1,1}, \mathit{error}, \}$$

and we depict this automaton to see its edges in the upper part of Figure 4.10.

Now, we replace all $\neg r'_{1,1}$ by the negated expression of $\hat{r}_{1,1}$ $[\![m]\!]$ original t-assignment, i.e, by $\neg m$, and all $r'_{1,1}$ are substituted by the reset operation of on delay timer i.e., by $m \wedge (\tau_1 := 0)$.

Read accesses to timer bit DN are all replaced by on timer condition $r_{1,1} \wedge (\tau_1 \geq c_1$, which yields $m \wedge (\tau_1 \geq 3)$. The results is depicted in the middle part of Figure 4.10.

We see that the timed automaton contains one edge, which is never be active in PLC, since it will not hold $m \wedge (\tau_1 \geq 3)$ in state 0, because m must go to 1 first to active clock. Removing this edge yields final timed automaton depicted in the bottom of Figure 4.10.

The example also demonstrates a disadvantage of the proposed conversion — the result is not optimal timed automaton. The automaton could include some redundant edges or states respectively, whose deletions are required. This task could have very high complexity.

The improvement of outlined conversion process has remained for a future research together with its deeper theoretical analysis.

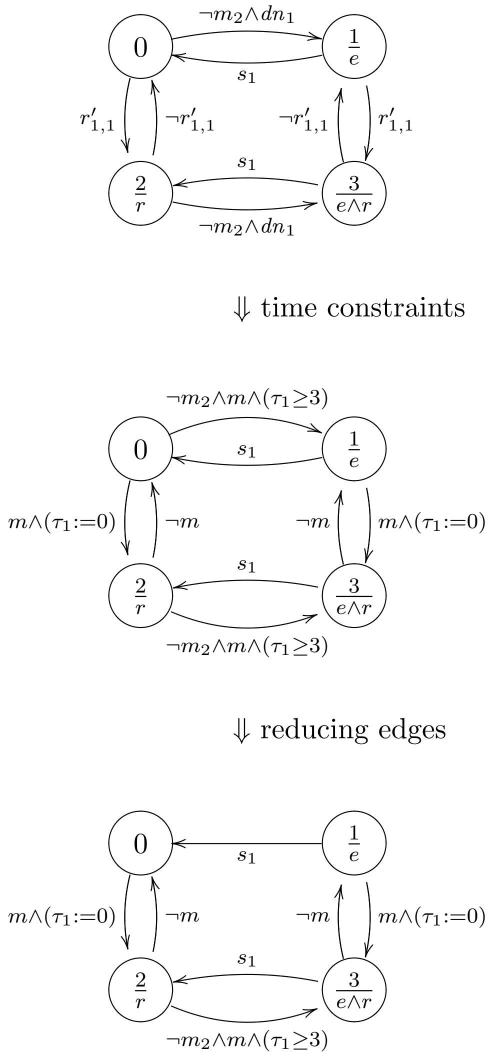


Figure 4.10: Creation of Time Automaton

Chapter 5

Future Work and Conclusion

Programmable logical controllers (PLCs) have proven their worth in countless industrial applications, since they were designed for high hardware reliability, hard environmental conditions, and efficient collecting I/O data.

But the safety of their software is an unknown element in general and the formal validation is needed. However, before verifying any PLC program we must convert its code into some universal language acceptable by chosen model checker. This conversion represents necessary step for any intended verification and its resolving opens possibilities for easier application of many known methods and tools during the development of PLC programs.

Therefore, PLC conversion was selected as the main challenge of this thesis. We present three contributions, which are probably novel approaches that have not been published yet:

- APLC machine for modelling wide range of PLC programs with bit instructions, jumps and subroutines was designed and its extension to timer instructions was demostrated. APLC machine forms a universal base for converting source codes of many different PLCs regardless of their manufacture.

- New theory of transfer sets was invented and the existence of the monoid of the transfer sets has been proved. The monoid allows associative composition of abstract PLC programs to logical formulas that are usually accepted by any model checker or tool in some form.

- APLCTRANS algorithm was designed that performs converting in linear time in the size of PLC source code at the most cases, though the converted program has an exponential complexity of its execution time.

The advantages of our methodology (together with its remaining imperfections) have been demonstrated by experimental results. We believe that our technique is general enough to be adapted for a wide range of PLCs.

There are many interesting avenues for future research and many lines of further development suggest themselves. Clearly, our work has revealed a lot of possibilities for improvements, extensions, and APLCTRANS applications. These have been already discussed in the relevant sections.

Here we list two new ideas of future development that have not been mentioned in the previous text:

- Abstract PLC does not convert arithmetic instructions. First theoretical studies show that APLCTRANS could deal with this problem with the aid of the conditional assignments.
- Some PLCs offer special sequential programming usually based on Grafcet. Converting these programs to either automata or Petri nets model with APLCTRANS cooperation could open possibilities for new modelling techniques.

In sum, our preliminary results are encouraging. We hope that further research along these and other lines will enable the potential advantages of APLCTRANS to apply to a much wider range of PLC programs.

But in spite of this progress, our thesis still remains a small contribution to 'hard' task of the formal verification, where are many hard problems waiting for efficient algorithms and much more of harder ones waiting for revealing.

Appendix A

Used Definitions

In this appendix, we present a raw collection of some basic definitions even if we assume that reader is familiar with them. They will serve us as an overview of notations and terminology which we refer to. We hope to resolve by them all possible ambiguities.

We denote an ordered set of integer numbers by I i.e., $I \overset{df}{=} \{1, 2, \ldots, |I|\}$.

Definition A.1 A binary relation R from a set X to a set Y *is an arbitrary subset of the cartesian product* $X \times Y$*. If* $X = Y$ *then* R *is called* a binary relation on the set X*.*

If a binary relation name has an alphabetic label, for instance R, then belonging x, y to R is usually written as $\langle x, z \rangle \in R$, but if a relation is denoted by a special symbol, for instance $\equiv$, that more readable form is used as $x \equiv y$.

Definition A.2 *A* function *or* mapping g *is a binary relation* $\mathrm{g} : X \to Y$ *from a set* X *to a set* Y *satisfying the following two properties:*

(functional) $\langle x, y \rangle \in \mathrm{g}$ *and* $\langle x, z \rangle \in \mathrm{g}$ *imply* $y = z$

(total) *for each* $x \in X$ *exists at least one* $y \in Y$ *such that* $\langle x, y \rangle \in \mathrm{g}$

The set X *of all admissible arguments is called the* domain of g *denoted by* $dom(\mathrm{g})$*. The set* Y *of all admissible values is called the* codomain of g *denoted by* $co(\mathrm{g})$*.*

Traditionally, functions are written as formulas that converted some input value (or values) into an output value. If g is the name for a function and x is a an input value, then $\mathrm{g}(x)$ denotes the output value corresponding to x under the rule g. We will always prefer this form to relation notation $\langle x, y \rangle \in \mathrm{g}$. An input value is also called an *argument* of the function, and an output value is called a *value* of the function.

In case of denoting a function with one or two arguments by a special non alphabetic symbol, for instance $\circ$ or $\perp$, such functions will be expressed as an operator. For example, instead of writing $z = \circ(x, y)$ or $z =\perp (x)$ where $x, y,$ and z are some variables, we will write $z = x \circ y$ or $z =\perp x$ and call $\perp$ as unary operator and $\circ$ as binary operator.

Proposition A.1 *Let* g *be a function with finite domain and codomain. If* $|dom(\mathrm{g})| = |cod(\mathrm{g})|$ *then* g *is bijective mapping from* $dom(\mathrm{g}) \rightarrow co(\mathrm{g})$.

Proof: Suppose that $x, y \in dom(\mathrm{g})), x \neq y$ have equal output values $\mathrm{g}(x) = \mathrm{g}(y)$ which satisfy $\mathrm{g}(x), \mathrm{g}(y) \in co(\mathrm{g})$. Because functions map each argument to exactly one output value (total and functional property), the sets $dom(\mathrm{g})$ and $co(\mathrm{g})$ cannot have the same cardinality. We have a contradiction. □

The bijective mappings are defined occasionally as those satisfying equality of the cardinalities of their domains and codomains.

Definition A.3 *A binary relation R on a set X is a* partial ordering *if for all $x, y, z \in X$:*
(reflexivity) $\langle x, x\rangle \in R$
(antisymmetry) $\langle x, y\rangle \in R$ *and* $\langle y, x\rangle \in R$ *imply* $x = y$
(transitivity) $\langle x, y\rangle \in R$ *and* $\langle y, z\rangle \in R$ *imply* $\langle x, z\rangle \in R$
Ordering R is called total ordering *if it satisfies: for every $x, y \in X$ we have* $\langle x, y\rangle \in R$ *and* $\langle y, x\rangle \in R$.

Definition A.4 *A binary relation R on a set X is called* equivalence relation *if for all $x, y, z \in X$:*
(reflexivity) $\langle x, x\rangle \in R$
(symmetry) $\langle x, y\rangle \in R$ *iff* $\langle y, x\rangle \in R$
(transitivity) $\langle x, y\rangle \in R$ *and* $\langle y, z\rangle \in R$ *imply* $\langle x, z\rangle \in R$
A set $\widetilde{R}(x) \overset{df}{=} \{y \in X \mid \langle x, y\rangle \in R\}$ *is called* equivalence class *corresponding to X and set of all equivalence classes* $X/R \overset{df}{=} \{\widetilde{R}(x) \mid x \in X\}$ *is called the* factor set.

Two *trivial equivalences* always exist on any set X. *Identical equivalence*

$$\Delta_X \overset{df}{=} \{\langle x, x\rangle \ \mid x \in X\} \tag{A.1}$$

and universal equivalence $\{\langle x, y\rangle \ \mid x, y \in X\}$. The other equivalences are called *non trivial equivalences.*

Definition A.5 *A set of nonempty subsets $\{X_i \mid i \in I\}$ of a set X is called* a decomposition of X *if* $X = \bigcup_{i\in I} X_i$ *and* $X_i \cap X_j = \emptyset$ *for all* $i, j \in I, \quad i \neq j$.

Proposition A.2 *Let R be an equivalence on a set X then the factor set $X/R = \{\widetilde{R}(x) \mid x \in X\}$ is a decomposition of the set X.*

Proof: For all $x \in R$ is $\widetilde{R}(x) \neq \emptyset$ because $x \in \widetilde{R}(x)$ (reflexivity of R) and therefore $X \subseteq \bigcup_{x \in X} \widetilde{R}(x)$. We will show that $\widetilde{R}(x) \cap \widetilde{R}(y) \neq \emptyset$ implies $\widetilde{R}(x) = \widetilde{R}(y)$. Let, on a contrary, be $z \in \widetilde{R}(x) \cap \widetilde{R}(y)$ then for all $v \in \widetilde{R}(x)$ it holds $\langle v, x\rangle \in R$ and also $\langle z, x\rangle \in R$, therefore $\langle v, z\rangle \in R$ (symmetry and transitivity). Applying transitivity we can derive from $\langle z, y\rangle \in R$ that $\langle v, y\rangle \in R$ i.e., $v \in \widetilde{R}(y)$. It implies that $\widetilde{R}(x) \subseteq \widetilde{R}(y)$ and according to symmetry of R also $\widetilde{R}(x) \supseteq \widetilde{R}(y)$, therefore $\widetilde{R}(x) = \widetilde{R}(y)$ and X/R is a decomposition of X. □

Definition A.6 *System of equivalences R_i, $i = I, |I| = n$ on set X is called* separating*, if it satisfies*

$$\bigcap_{i=1}^{n} R_i = \{\langle x, x\rangle \mid x \in X\} \tag{A.2}$$

Definition A.7 (Semigroup) *A semigroup is a pair $(M, \odot)$, where M is a set and $\odot$ is a binary operation on M, obeying the following rules:*

(closure)	$a, b \in M$ *implies* $a \odot b \in M$
(associativity)	*for all* $a, b, c \in M$ *that holds* $(a \odot b) \odot c = a \odot (b \odot c)$

Definition A.8 (Monoid) *A monoid is a semigroup $(M, \odot)$ with an identity element e that satisfies the following rule:*

(identity)	*for all* $a \in M$, $a \odot e = e \odot a = a$

Definition A.9 *A lattice is a set L together with two binary operations $\wedge$ and $\vee$ such that for any $a, b, c \in L$*

(idempotency)	$a \vee a = a$	$a \wedge a = a$
(commutativity)	$a \vee b = b \vee a$	$a \wedge b = b \wedge a$
(associativity)	$a \vee (b \vee c) = (a \vee b) \vee c$	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$
(absorption)	$a \vee (a \wedge b) = a$	$a \wedge (a \vee b) = a$

Definition A.10 *A Boolean algebra is a lattice $(A, \vee, \wedge)$ with the following four additional properties:*

(bounded below)	$\exists\, 0 \in A$	$a \vee 0 = a \quad \forall a \in A$
(bounded above)	$\exists\, 1 \in A$	$a \wedge 1 = a \quad \forall a \in A$
(distributive law)	$\forall a, b, c \in A$	$(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$
(complements)	$\forall a \in A\ \exists \neg a \in A$	$a \vee \neg a = 1$ *and* $a \wedge \neg a = 0$

It directly follows from axioms above that the smallest element 0 and the largest element 1 are unique and every element has only one complement.

Definition A.11 (Kleene-closure) *Let E be a finite set of elements. Let us denote by E^* the set of all finite strings of elements of E, including empty string ε; the operation $()^*$ is called* Kleene-closure*. The set E is called* alphabet *and any subset $L \subset E^*$ is called a* language defined over alphabet *E.*

Observe that the set E^* is countably infinite whenever E is at most countable, since it contains strings of arbitrarily long length. For example, if $E = \{0, 1\}$, then its Kleene-closure $\{0, 1\}^*$ is the set with elements $E^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. The language without empty string is usually denoted by $E^+ \overset{df}{=} E^* - \{\varepsilon\}$.

Definition A.12 (Concatenation of strings) *Let L be a language $L \subseteq E^*$ and $s, t \in L$ two strings. Let us denote by . the binary operation defined on E^* that performs the concatenation of strings, $s.t$ is the new string consisting of the elements s immediately followed by the elements of in t.*

The empty string ε is the identity element of concatenation: $s.\varepsilon = \varepsilon.s = s$ for any string s.

Definition A.13 *Let L be a language $L \subseteq E^*$ and $s \in L$. If $s = t.u.v$ where . denote the operation of concatenation of strings then:*

- *t is called a* prefix*,*
- *u is called a* substring*, and*
- *v is called a* suffix *of s.*

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFS98] Alexander Aiken, Manuel Fähndrich, and Zhendong Su. Detecting races in relay ladder logic programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 4th International Conference, TACAS'98*, pages 184–200, 1998.

[AH97] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams. In *LICS, IEEE Symposium on Logic in Computer Science*, 1997.

[AJ98] P. Aziz Abdulla and B. Jonsson. Verifying networks of timed processes. *Lecture Notes in Computer Science*, 1384:298–312, 1998.

[Alu00] Rajeev Alur. *Verification of Digital and Hybrid Systems*, chapter Timed Automata, pages 233–264. Springer-Verlag Berlin, 2000.

[And94] Mark Andrews. *C++ Windows NT Programming.* MT Books, 1994.

[And97] Henrik Reif Andersen. An introduction to binary decision diagrams. Technical report, Department of Information Technology, Technical University of Denmark, October 1997.

[AT98a] Stuart Anderson and Konstantinos Tourlas. Design for proof: An approach to the design of domain-specific languages. In *The Third FMICS Workshop*, May 1998.

[AT98b] Stuart Anderson and Konstantinos Tourlas. Diagrams and programming languages for programmable controllers. *Formal Aspect of Computing*, 10 (5-6):452–468, 1998.

[BBF$^+$01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification.* Springer, 2001.

[BHLL00] Sebastien Bornot, Ralf Huuck, Ben Lukoschus, and Yassine Lakhnech. Utilizing static analysis for programmable logic controllers. In *4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems*, September 18–19 2000.

[BHŠ00] Jiří Bayer, Zdeněk Hanzálek, and Richard Šusta. *Logical systems for control engineering.* CTU-FEE Prague, 2000. In Czeck lang.

[BM00] Ed Brinksma and Angelika Mader. Verification and optimization of a PLC control schedule. In *7th SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science.* Springer Verlag, 2000.

[BMPY97] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Computer-Aided Verification, CAV'97, Israel*, Lecture Notes in Computer Science. Springer-Verlang, 1997.

[BS90] Ravi B. Boppana and Michael Sipser. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, chapter The Complexity of Finite Functions, pages 757–804. Elsevier and MIT Press, 1990.

[CHB01] Colin Chambers, Mike Holcombe, and Judith Barnard. Introducing X-machine models to verify PLC ladder diagrams. *Computers in Industry*, 45:277–290, July 2001.

[CL99] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[Com] International Electrotechnical Commission. *International Standard 1131, Programmable Controllers. Part 3: Programming Languages.* 1993.

[DCR$^+$00] O. De Smet, S. Couffin, O. Rossi, G. Canet, J.-J. Lesage, Ph. Schnoebelen, and H. Papini. Safe programming of PLC using formal verification methods. In *Proc. 4th Int. PLCopen Conf. on Industrial Control Programming (ICP'2000), Utrecht, The Netherlands, Oct. 2000*, pages 73–78. PLCOpen, Zaltbommel, The Netherlands, 2000.

[DFMV98] Henzing Dierks, Ansgar Fehnker, Angelika Mader, and Frits Vaandrager. Operational and logical semantics for polling real-time systems. In *FTRTFT'98.* Springer Verlag, 1998.

[DFT01] Henning Dierks, Hans Fleischhack, and Josef Tapken. *Moby/PLC tutorial.* Departments of Theoretical Informatics, Oldenburg, 2001.

[Die97] Henning Dierks. Synthesizing controllers from real-time specifications. In *Tenth International Symposium on System Synthesis (ISSS '97)*, pages 126–133. IEEE Computer Society Press, 1997.

[Die00] Henning Dierks. *Specification and Verification of Polling Real-Time Systems.* PhD thesis, Carl-von-Ossietzky Universität Oldenburg, 2000.

[DK90] Marie Demlová and Václav Koubek. *Algebraic theory of automata.* SNTL Praha, 1990. Printed only in Czech language.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. *The Tool Kronos*, chapter 0, page 0. Springer-Verlag, lecture notes in computer science 1066 edition, 1996.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *IEEE Real-Time Systems Symposium, RTSS'96, Washington, DC, USA*. IEEE Computer Society Press, 1996.

[FH01a] Petr Fišer and Jan Hlavička. BOOM - a heuristic boolean minimizer. In *International Conference on Computer-Aided Design ICCAD 2001, San Jose, California (USA)*, pages 439–442, 2001.

[FH01b] Petr Fišer and Jan Hlavička. A heuristic method of two-level logic synthesis. In *The 5th World Multiconference on Systemics, Cybernetics and Informatics SCI'2001, Orlando, Florida (USA)*, volume II, pages 283–288, 2001.

[FH01c] Petr Fišer and Jan Hlavička. Implicant expansion method used in the BOOM minimizer. In *IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'01), Gyor (Hungary)*, pages 291–298, 2001.

[FH01d] Petr Fiser and Jan Hlavicka. On the use of mutations in boolean minimization. In *Euromicro Symposium on Digital Systems Design, Warsaw*, pages 300–305, 2001.

[Fit90] John S. Fitzgerald. *Case Studies in Systematic Software Development*, chapter 5, pages 127–162. Prentice-Hall International, 1990.

[Fra92] Nissim Francez. *Program Verification.* Addison-Wesley Publishers Ltd, 1992.

[FŠ92] Jindřich Fuka and Richard Šusta. Processing visual information in industrial control with the aid of M module. *Automatization*, 9:263–266, 1992. In Czech lang.

[Gra96] Paul Graham. *ANSI Common Lisp.* Prentice-Hall, 1996.

[Grö99] Clemens Gröpl. *Binary Decision Diagrams for Random Boolean Functions.* PhD thesis, Humboldt Universite Berlin, 1999.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages, Structures and Techniques.* The MIT Press, Massachusetts Institute of Tecnology, 1992.

[HD93] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th ACM/IEEE Design Automation Conference, Dallas, TX, USA*, pages 266–271, June 1993.

[Hol97] Gerard J. Holzmann. *Basic Spin Manual.* Bell Laboratories, 1997.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F.* Springer-Verlag, 1985.

[Hug89] Thomas A. Hughes. *Programmable Controlles.* Instrument Society of America, 1989.

[HWA97] Henrik Hulgaard, Poul Frederick Williams, and Henrik Reif Andersen. Combinational logic-level verification using boolean expression diagrams. In *3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1997.

[HWA99] Henrik Hulgaard, Poul Frederick Williams, and Henrik Reif Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions of Computer-Aided Design, 18(7)*, July 1999.

[Int92] International Electrotechnical Commision. *IEC 848 Grafcet*, 1992.

[LLW95] François Laroussinie, Kim G. Larsen, and Carsten Weise. From timed automata to logic - and back. Technical Report RS-95-2, BRICS, jan 1995. 21 pp. Accessible through url: http://www.brics.aau.dk/BRICS.

[LPW97] Kim Guldstrand Larsen, Paul Pettersson, and Yi Wang. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfe*, 0(1 (1+2)):134–152, September 1997. 0.

[LT93] Nancy Leveson and Clark S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[Mad00a] Angelika Mader. A classification of PLC models and applications. In *WODES 2000: 5th Workshop on Discrete Event Systems, Gent, Belgium*, August 21–23 2000.

[Mad00b] Angelika Mader. Precise timing analysis of PLC applications two small examples. Technical report, University of Nijmegen, 2000.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking, An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, School of Computer Scienc, 1993.

[McM97] K. L. McMillan. *Getting started with SMV.* Cadence Berkeley Labs, 1997. Document available at URL: http://www-cad.eecs.berkeley.edu/ kenmcmil/.

[McM00] Kenneth L. McMillan. *Verification of Digital and Hybrid Systems*, chapter Compositional Systems and Methods, pages 138–151. Springer-Verlag Berlin, 2000.

[Min99] Mark Minas. Creating semantic representations of diagrams. In *AGTIVE*, pages 209–224, 1999.

[MLAH01] Jesper Moeller, Jakob Lichtenberg, Henrik Andersen, and Henrik Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In Alessandro Cimatti and Orna Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 23. Elsevier Science Publishers, 2001.

[MW99] Angelika Mader and Hanno Wupper. Timed automaton models for simple programmable logic controllers. In *Proceedings of Euromicro Conference on Real-Time Systems 1999, York, UK*, 1999.

[MW00] Angelika Mader and Hanno Wupper. What is the method in applying formal methods to plc applications? In *ADPM 2000*, 2000.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, A Formal Introduction.* John Wiley and Sons, 1999.

[Nov01] Radim Novotný. *Distributed Control Systems.* PhD thesis, Faculty of Electrical Engineering, Prague, 2001. In Czech lang.

[OCVSV98] Arlindo L. Oliveira, Luca P. Carloni, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. Exact minimization of binary decision diagrams using implicit techniques. *IEEE Transactions on Computers*, 47(11), 1998.

[OD98] Ernst-Rudiger Olderog and Henning Dierks. Decomposing real-time specifications. *Lecture Notes in Computer Science*, 1536:465–489, 1998.

[Pel00] Doron Peled. *Verification of Digital and Hybrid Systems*, chapter Model Checking Using Automata Theory, pages 55–79. Springer-Verlag Berlin, 2000.

[Ram99] John D. Ramsdell. The tail-recursive Machine. *Journal of Automated Reasoning*, 23(1):43–62, 1999.

[RK98] M. Rausch and B. H. Krogh. Formal verification of PLC programs. In *American Control Conference, Philadelphia, PA, USA*, 1998.

[Roc98] Rockwell Automation. *PLC 5 Programmable Controllers, Instruction set reference*, 1998. On-line manual available at URL: http://www.ab.com/manuals/cp/.

[Roc01a] Rockwell Automation. *Logix5000 Controllers Common Procedures, Publication 1756-PM001D-EN-P*, 2001. On-line manual available at URL: http://www.ab.com/manuals/cl/.

[Roc01b] Rockwell Automation. *SLC500 Instruction Set, Reference Manual. Publication 1747-RM001C-EN-P*, 2001. On-line manual available at URL: http://www.ab.com/manuals/cp/.

[RS00] O. Rossi and Ph. Schnoebelen. Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In *Proc. 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM'2000), Dortmund, Germany, Sept. 2000*, pages 177–182. Shaker Verlag, Aachen, Germany, 2000.

[Sie02] Siemens AG. *S7-200 Programmable Controller System Manual Edition*, 2002. System Manual Order number: 6ES7298-8FA22-8BH0.

[Slo94] M. Sloman. *Distributed Control.* Prentice Hall, 1994.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, *Proceedings 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FM-CAD'98, Palo Alto, CA, USA, 4–6 Nov 1998*, volume 1522, pages 82–99. Springer-Verlag, Berlin, 1998.

[SSL$^+$92] M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: system for sequential circuit synthesis. Technical report, 1992.

[Šus93] Richard Šusta. Visual information in industrial control systems. In *Informatics in CIM, Prague*, 1993.

[Šus99] Richard Šusta. *Programming for Control Engineering in Windows.* CTU-FEE Prague, 1999. In Czeck lang.

[Šus02] Richard Šusta. Paralel abstraction of PLC program. In *The 5th International Scientific -Technical Conference Process Control 2002, Říp 2002*, June 2002. CD R058 1-13.

[Šus03] Richard Šusta. APLCTRANS algorithm for PLC verification. In *14th International Conference Process Control 2003, Štrbské Pleso, Slovakia*, June 2003. Proceedings ISBN 80-227-1902-1, CD-ROM.

[SV96] Springintveld and Vaandrager. Minimizable timed automata. In *S: Formal Techniques in Real-Time and Fault-Tolerant Systems: International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS.* LNCS, Springer-Verlag, 1996.

[SVD97] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. Technical Report CSI-R9712, University of Nijmegen, Computing Science Institute, 1997.

[TD98] Josef Tapken and Henning Dierks. Moby/PLC - graphical development of PLC-automata. In A.P. Ravn and H. Rischel, editors, *Proceedings of FTRTFT'98*, volume 1486 of LNCS, pages 311–314. Springer Verlag, 1998.

[Tou97] Konstantinos Tourlas. An assessment of the EC 1131–3 standard on languages for programmable controllers. In Peter Daniel, editor, *SAFECOMP97: the 16th International Conference on Computer Safety, Reliability and Security York,* , pages 210–219. Springer, September 1997.

[Tou00] Konstantinos Tourlas. *Diagrammatic Representations in Domain-Specific Languages.* PhD thesis, Univesity of Edinburgh, 2000.

[Wil00] Poul Frederick Williams. *Formal Verification Based on Boolean Expression Diagrams.* PhD thesis, DTU Tryk, Lyngby, 2000.

[YBO$^+$98] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. A performance study of BDD-based model checking. In *Proc. of the Formal Methods on Computer-Aided Design (November 1998)*, pages 255–289, 1998.

[Yov98] S. Yovine. *Embedded Systems*, chapter Model-checking Timed Automata, page 0. Lecture Notes in Computer Science 1494. 0, 1998.

Index