RWTH Aachen

Department of Computer Science
Technical Report

Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

Sebastian Biallas

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2016-07

RWTH Aachen · Department of Computer Science · July 2016

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Sebastian Biallas
aus Düsseldorf

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr.-Ing. Alexander Fay

Tag der mündlichen Prüfung: 14.7.2016

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über `http://dnb.d-nb.de` abrufbar.

Zugl.: D 82 (Diss. RWTH Aachen University, 2016)

Sebastian Biallas
Informatik 11 — Embedded Software
`biallas@embedded.rwth-aachen.de`

Aachener Informatik Bericht AIB-2016-07

Herausgeber:	Fachgruppe Informatik RWTH Aachen University Ahornstr. 55 52074 Aachen GERMANY

ISSN 0935-3232

Printed in Germany.

ISBN 978-3-8440-4711-0

Shaker Verlag GmbH • Postfach 101818 • 52018 Aachen
Telefon: 02407 / 95 96 - 0 • Telefax: 02407 / 95 96 - 9
Internet: `www.shaker.de` • E-Mail: `info@shaker.de`

ABSTRACT

Programmable Logic Controllers (PLCs, ger. Speicherprogrammierbare Steuerungen) are control devices used in industry to control, operate and monitor plants, machines and robots. PLCs comprise input connectors, typically connected to sensors, output connectors, typically connected to actuators, and a program, which controls the behavior, computing new output values based on the input values and an internal state. Since PLCs operate in safety-critical environments, where a malfunction could seriously harm the environment, humans, or the plant, it is recommended to verify their programs using formal methods.

This dissertation studies the formal methods *model checking* and *static analysis* to prove the correctness of PLC programs. For this, we developed the tool Arcade.PLC, which allows the automatic application of these methods to PLC programs written in different vendor-specific dialects. It extracts a model from the program by abstract simulation, which over-approximates the possible program behavior. The user is then able to verify whether the model obeys a specification, which can be written in the logic CTL or using automata.

For applying model checking, we demonstrate how the model can be extracted automatically, such that the approach scales to industrial size programs. For this, we introduce two different abstraction techniques: First, we develop an abstraction refinement guided by the model checker that automatically creates an abstracted model by iteratively analyzing counterexamples. Second, we implemented a predicate abstraction that evaluates a formalized program semantics using an SMT solver. Both techniques are evaluated using different programs from industry and academia. Further, we introduce a simplified formalism to write specifications, which is influenced by an automata-based language established in industry. We implement an algorithm to check programs using this formalism and show that this technique is even able to detect errors in the specification. Finally, we detail how counterexamples generated by the model checker can be analyzed automatically to locate the actual erroneous line in the program.

The static analysis we developed is able to detect program errors in a fully automatic way. We detect typical errors such as division by zero and illegal array accesses, but also PLC specific errors, e. g., during a restart. The analysis is based on a value-set analysis, which determines the values of all program variables in each program location. These sets are then verified against the predefined checks or user-provided annotations. We show how to implement this analysis such that it scales to industrial size programs. The approach is evaluated on an industrial case study.

Speicherprogrammierbare Steuerungen (SPSen, engl. Programmable Logic Controller) sind Automatisierungsgeräte, welche zur Steuerung, Regelung und Überwachung von industriellen Anlagen und Maschinen eingesetzt werden. Sie besitzen dazu Eingänge, die mit Sensoren verbunden sind, Ausgänge, die mit Aktuatoren verbunden sind und ein Programm, welches die Ausgänge in Abhängigkeit der Eingänge und eines internen Speichers belegt. Da SPSen häufig in kritischen Bereichen eingesetzt werden, in denen eine Fehlfunktion Gefahren für Mensch, Umwelt oder die Anlage bergen kann, ist die Korrektheit des Programms zu prüfen.

Diese Dissertation untersucht die formalen Methoden *Model-Checking* und *Statische Analyse*, um die Korrektheit von PLC-Programmen zu beweisen. Wir haben dazu das Tool ARCADE.PLC geschrieben, welches es ermöglicht, diese Techniken vollautomatisch auf PLC-Programme verschiedener Hersteller anzuwenden. Es extrahiert durch abstrakte Simulation ein Modell, welches sämtliches Programmverhalten widerspiegelt. Der Benutzer kann dann überprüfen, ob das Modell einer Spezifikation entspricht, welche er in der Logik CTL formulieren muss oder als Automaten eingegeben kann.

Zum Bereich Model-Checking zeigen wir in dieser Dissertation, wie das Modell automatisch abstrahiert werden kann, so dass der Ansatz auch für industrielle Programme skaliert. Es werden dazu zwei verschiedene Abstraktionstechniken eingeführt: Eine durch den Model-Checker gesteuerte Abstraktionsverfeinerung erstellt ein abstrahiertes Modell iterativ durch Analyse von Gegenbeispielen. Außerdem haben wir eine automatische Prädikat-Abstraktion implementiert, welche mithilfe einer SMT-Solvers die formalisierte Programmsematik auf Prädikaten auswertet. Beide Techniken werden anhand verschiedener Programme evaluiert. Weiterhin führen wir einen vereinfachten Spezifkationsformalismus ein, welcher sich an einer in der Industrie etablierten Automatensprache orientiert. Wir implementieren einen Algorithmus, um Programme mit diesem Formalismus zu überprüfen und zeigen, dass durch diese Technik auch Spezifikationsfehler entdeckt werden können. Schließlich zeigen wir noch, wie vom Model-Checker gefundene Gegenbeispiele analysiert werden können, um die eigentlich fehlerhafte Programmzeile automatisch zu lokalisieren.

Die von uns implementierte Statische Analyse kann vollautomatisch Programmfehler entdecken. Dazu gehören beispielsweise eine Division durch Null, unerlaubte Array-Zugriffe oder PLC-spezifische Fehler z.B. beim Neustart. Die Analyse basiert auf einer Wertebereichsanalyse, welche eine Übermenge der Werte aller Variablen in allen Programmstellen berechnet. Wir zeigen, wie diese Analyse skalierbar implementiert werden kann. Der Ansatz wird an einer großen industriellen Fallstudie ausgewertet.

ACKNOWLEDGEMENTS

I worked on this dissertation while I was employed as a research assistant at the chair *Informatik 11 — Embedded Software* at the RWTH Aachen University. This work would not have been possible without the support of many others. First of all, I have to thank Prof. Dr.-Ing. Stefan Kowalewski for giving me the opportunity to join his group, for supporting my thesis, and for the excellent collaboration during this time. I would also like to thank Prof. Dr.-Ing. Alexander Fay for serving as a second supervisor and for helpful remarks. Furthermore, I thank Prof. Dr. Bernhard Rumpe and apl. Prof. Dr. Thomas Noll for participating in my examination committee.

I have to thank Dr. Bastian Schlich, whom I first met when I was a student while he was a postdoctoral researcher at the chair. He introduced me to the topic of formal verification of embedded software and supervised by Diploma thesis. Later, he went to ABB and was able to establish an industrial research collaboration. I would also like to thank his colleagues, especially Dr. Stefan Hauck-Stattelmann, for the great collaboration during this time.

Furthermore, I would like to thank all my former colleagues and friends for the great time I had at the chair. In particular, I enjoyed the numerous interesting discussions I had with Dr. Jörg Brauer even outside the work environment.

I also have to thank Dr. Ralf Huuck for giving me the possibility to join his group at NICTA during a research visit. I learned a lot about the static analysis of C and C++ programs and how to scale such an analysis to industrial size programs.

I especially have to thank my students. They wrote excellent bachelor and master theses, implemented algorithms and user interfaces, performed case studies, and contributed as co-authors of publications. This work would not have been possible without them.

Financially, my work was supported by the *Deutsche Forschungsgemeinschaft*. Further, I was supported by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* and the DFG Cluster of Excellence on *Ultra High Speed Mobile Information and Communication*. I am very grateful for this support and I also have to thank these groups for many interesting and stimulating discussions.

Finally, I have to thank my parents and my sister for support and proof-reading.

Sebastian Biallas
July 2016, Berlin

CONTENTS

1 INTRODUCTION

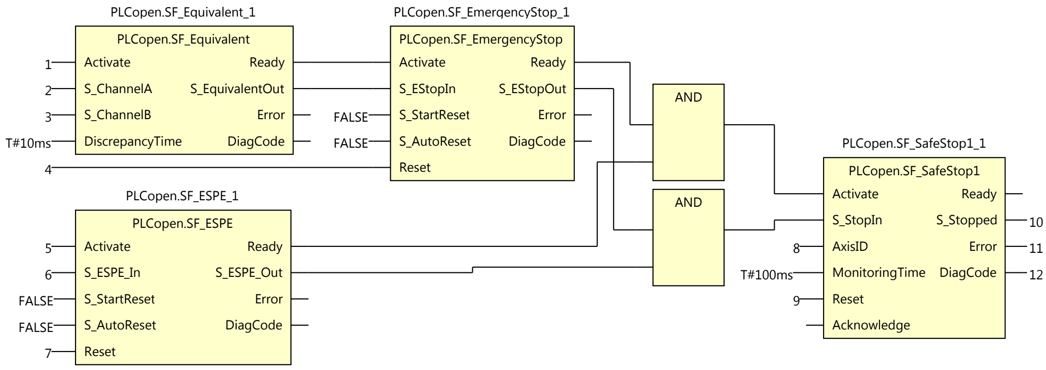
Programmable Logic Controllers (PLCs) are control devices used in industry to control, operate, supervise, and monitor machines, robots, assembly lines, chemical plants, power plants, oil rigs, and other technical processes [71]. They typically comprise a set of input/output signals, which are connected to sensors and actuators, and a program. The program is then run periodically at a high frequency to calculate new output values based on the current input values and the internal memory [71]. Since PLCs are often used in safety-critical settings, where a failure might cause serious harm to humans or the environment, the correctness of the program is highly important. Hence, its functionality must undergo an extensive testing process [75, 96].

From a practical standpoint, however, testing cannot test every possible combination and sequence of input signals, especially when the internal state of the PLC program has to be considered as well. Thus, testing is restricted to certain coverage criteria [91]. Formal verification, on the other hand, strives to validate that a property holds in *every* possible configuration of the program. This is achieved by reasoning about a formal model of the program behavior. In model checking [4], e. g., the program behavior is modeled as a transition system, which reflects the reachable states of the system and their transitions. It is then checked whether this model satisfies properties given in modal or temporal logics (cp. Sect. 2.4). This approach is additionally able to generate a counterexample in case of a violation. The counterexample then explains the violation by stating the exact circumstances in which a property is violated, which aids in debugging the problem. Another type of formal methods are static analyses, which can be used without a user supplied specification. They are able to detect typical programming errors such as unreachable code, possible divisions by zero, index out-of-bounds accesses, variable stuck-to-zero. In some cases, the careful application of formal methods can even reveal errors in the specification (cp. Sect. 6.5 and 8.6.3).

Model Checking

Static Analyses

Formal verification is often desirable, if not recommended [70], when dealing with safety-critical code. The goal of this dissertation is to harness the formal methods model checking and static analysis for the verification of PLC code. The emphasis is to adapt and refine these formal methods to the PLC domain to make them applicable to industrial code and usable by engineers.



1	`InputDevice1_active`	2	`S1_S_EStopIn_1`
3	`S1_S_EStopIn_2`	4	`S0_RESET`
5	`InputDevice2_active`	6	`S2_S_ESPEIn`
7	`S0_RESET`	8	`AxisID_1`
9	`S3_DriveReset`	10	`S_Stopped`
11	`Error`	12	`DiagCode`

Figure 1: Example program taken from [94, Part 2, p. 19] and remodeled in our own tool for verification purpose.

1.1 FORMAL VERIFICATION OF PLC CODE

PLC programs are usually composed in a modular fashion (cp. Sect. 2.2.5). Figure 1 shows such a program, which is written in the graphical language *Function Block Diagram*. The example connects existing functions blocks using logical blocks to form a safety function. The goal of this safety function is to supervise an emergency stop button (inputs 2 and 3) and a light curtain (input 5) in order to stop a motor if any of these devices is triggered. Additionally, it is ensured that the motor can only be restarted if a manual reset is triggered after a safe stop was requested. The two redundant signals `S1_S_EStopin_1` and `S1_S_EStopin_2` of the emergency stop button are combined using an `SF_Equivalent` block, which allows for a certain discrepancy time that ensures to keep a consistent signal if both switches do not react at the same instant. The function blocks used in this diagram are defined by the PLCopen consortium [94, Part 1] and then provided by a vendor. Such a setup, i. e., combining existing function blocks from a library by an engineer, is typical for safety functions.

To assess and verify the correctness of such a program, we address three questions in this dissertation:

1. Is the safety function of this program implemented correctly?
2. Are the function blocks used in this program implemented correctly?
3. How can we automatically detect faulty PLC code?

We created the tool Arcade.PLC, which allows to apply formal verification to a wide range of industrial PLC code. It offers the formal method model checking,

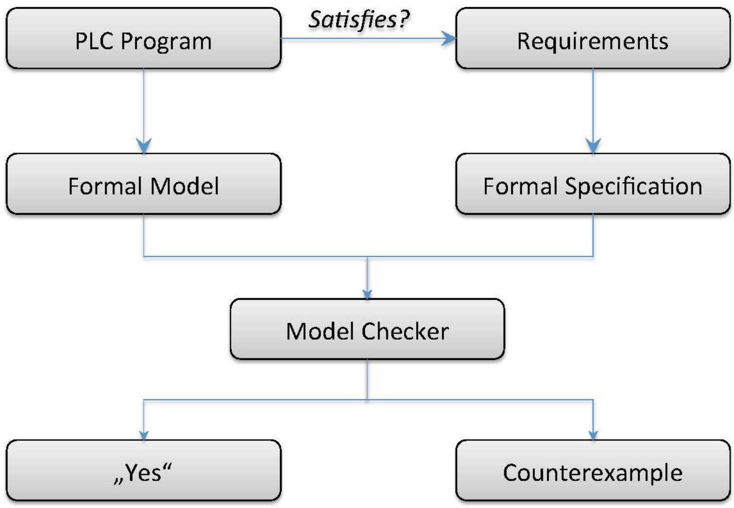


Figure 2: The model checking process.

which—as we will show—can be used to answer the first two questions. It also implements different static analyses to address the third question. We detail our contributions to these fields in the next section.

1.2 CONTRIBUTION & OUTLINE

We begin by describing the semantics of PLC programs in Chap. 2. We then derive a formal model, on which our verification methods are based on. This formal model comprises a transition system reflecting the observable input/output behavior of a PLC program or a function block. To automatically generate this model from a given PLC program and to check this model, we created the tool ARCADE.PLC. The history and overall organization of ARCADE.PLC is described in Chap. 3. It contains different parsers that translate PLC programs into an intermediate representation so as to handle multiple function blocks written in different languages in one framework.

1.2.1 *Model Checking*

In ARCADE.PLC, we use model checking to verify that a PLC program satisfies its requirements. An overview of the model checking process is depicted in Fig. 2: Given a *program* and *requirements*, the user wants to know whether the program satisfies the requirements. To apply formal methods, both, the program and the requirements have to be formalized first. Therefore, the program is transformed in a formal model, that reflects the possible program behavior. The requirements have to be formalized as well. Typically, logical formulae are used to specify the formal requirements. Then, a model checker is used to verify that the formal model satisfies the formal specification. If the model checker can prove that the model satisfies the requirements, it answers "yes"; otherwise a counterexample can be generated, which explains the violation by listing a sequence of events possible in the model that are forbidden by the specification. As a third option, the model checker might return "out of memory" or it might have to be interrupted prematurely, because it could not produce a result in a reasonable time. This happens if the model is too large to be explored by the model checker and necessitates abstractions to generate succinct models.

We present the following contributions to the model checking process of PLC programs:

- After an introduction to PLCs and their semantics in Chap. 2, we describe a formal model for the execution of PLC programs in Sect. 2.5.1. We further describe how this model can be generated automatically from a given program. This formal model takes the cyclic execution of PLC programs into account and can reflect a composition of different modules written in different languages.

- Since PLC programs usually depend on multiple inputs, *abstractions* of the model are necessary to make the approach feasible. We present an abstract model for PLC programs in Sect. 2.5.2.

- In Chap. 3 we give an overview of ARCADE.PLC and how it can be used as a model checker to automatically verify PLC programs.

- To derive abstractions automatically, we present a counterexample-guided abstraction refinement [40] scheme in Chap. 4, which we tailor for the PLC domain. This approach is evaluated using different programs from industry and academia in Sect. 4.6. In particular, it will be possible to verify properties of the function blocks shown in Fig. 1.

- To handle more complex programs, we describe a more powerful abstraction using an automatic solver in Chap. 5. This approach can abstract program states using all kinds of predicates (e. g., $var1 < var2$) that are expressible using the solver and thus handle more complex programs.

- To ease the formalization of program requirements, we introduce a specification formalism based on automata in Chap. 6. These automata represent a more natural expression of typical function block specifications. Since they are inspired by a formalism used in industry [55], many industrial specifications can readily be verified. We describe a model checking algorithm, which is evaluated on an industrial library.

- A model checker generates counterexamples for violations of all-quantified properties, which are concrete sequences of PLC input stimuli to trigger a violation. In Chap. 7 we present different heuristics that can localize the faulty program statement in a counterexample automatically, or at least reduce the number of possible error candidates and thus further aid the user in understanding a violation.

1.2.2 *Static Analysis*

Further, we implement different *Static Analyses* to derive program properties directly, i. e., without having to specify requirements[1]. The properties derived this way comprise the value-sets that the program variables can assume (at each program location or only input/output positions), positions where variables are dead or alive, summaries for function blocks, and pointer aliases. On the one hand, this information can then be used in other analyses, either to make them more precise or to speed them up. On the other hand, these techniques can reveal errors in the program without any specification effort from the user.

In particular, we present the following contributions in Chap. 8:

- In Sect. 8.2, we describe the static analysis process.
- In Sect. 8.2.4, we present a live and dead variable analysis, which takes the different calling conventions of PLC function blocks into account. We apply the dead variable analysis to make model checking PLC programs more efficient by reducing the state space sizes in Sect. 8.2.6.
- We present a pointer analysis to handle PLC programs using pointers.
- We present an efficient, flow-sensitive, partly context-sensitive abstract interpretation [44] of PLC programs. This analysis is based on the liveness information to make the approach feasible for large programs with a huge number of variables and function blocks.
- We use the information to implement a value set analyses, which compute a summary of the program behavior.
- Finally, the results of the static analysis are used to detect common programming mistakes. We are also able to check user-supplied assertions, which can directly be written in the source code. Results of an industrial case study are presented in Sect. 8.6.

1 Some authors also regard model checking as a form of static analysis as opposed to dynamic analyses that actually execute the program [122]. Other authors observed that the automatic extraction of the model using simulation can be seen as a dynamic analysis [90]. In this dissertation, we use the distinction that model checking requires a user-supplied specification, whereas static analysis can be run without manual specification effort.

1.2.3 *Combining Model Checking and Static Analysis*

Finally, we combine model checking techniques and static analysis techniques in Chap. 9. On the one hand, this provides even more powerful abstractions for the model checker, and allows us to verify the safety function of the program shown in Fig. 1. On the other hand, the model checker can be used to augment the results of the results of certain static analyses. The thesis concludes with an overview of the techniques presented and a discussion of their applicability in practice in Chap. 10.

1.3 RELATED WORK

The verification of PLC programs using formal methods has been extensively researched in the past [9, 86, 36, 8]. Moon [88] was the first to apply model checking as a verification technique of PLC programs written in Ladder Diagram. In the core of this approach, PLC programs are rewritten as an input language to the existing model checking tool SMV, which was used for the verification. A similar approach is followed by Pavlovic et al. [93]. They convert STEP7 Instruction List programs into inputs for the NuSVM model checker. Gourcuff et al. present an approach for Structured Text programs [62]. They only support Boolean variables and no control structures. In [63] they introduce an abstraction to make the approach feasible for larger programs. Mertke and Frey [85] first rewrite PLC programs into Petri nets, which are then analyzed using a model checker.

The verification of PLCopen function blocks was also discussed by Soliman and Frey [114]. They use the Uppaal model checker to verify compositions of multiple PLCopen function blocks. In their work, the focus is not on the correct implementation of a function block, but on how programs composed of PLCopen function blocks can be verified (presumed that the function block implementation is correct).

A different approach is presented by Süflow and Drechsler [116]. They use a SAT solver for equivalence checking of PLC programs using an intermediate representation in SystemC.

Recently, Darvas et al. [49] came up with an efficient method of verifying complex Structured Text programs using NuSMV. Their approach relies heavily on rewriting and simplifying the NuSMV models to make the technique efficient.

Ljungkrantz et at. [82] study formal methods to verify safety function blocks as well. They came up with ST-LTL to formulate their specifications, which also allows for past-time operators to ease the specification efforts. The work presented in this dissertation additionally researches automata based model checking and static analysis.

Šusta [118] also presents a verification framework for PLC programs operating on binary variables. His approach also depends on an intermediate representation (called APLC in his work) for representing the PLC program.

Bornot et al. [33] were the first to describe static analysis techniques for PLC programs. They use an abstract interpretation framework [44] similar to ours but are restricted to Instruction List programs. Prähofer et al. [95] give an overview about different static code analysis techniques and their benefits to IEC 61131-3 programs. Their approach is concerned with detecting bad programming practices (naming conventions, program complexity, code smells, dead locks) while the approach used in our work infers the possible values of all program variables to detect programming errors. They also assess available commercial tools for static PLC code analysis, which, at the moment, seem to focus on syntactic checks only, e.g., the compliance with certain naming convention for variables.

Direct model checking of Instruction List programs was introduced by Schlich et al. [105]. This approach, which does not necessitate the rewriting of the programs into other model checker inputs or Petri nets, allows for the verification of larger programs. The work presented in this dissertation continues this line of research by providing domain specific abstraction techniques, new specification formalisms, error localization techniques and a static analysis for industrial size programs. We present a more in-depth discussion of related work compared to our contributions in their respective chapters.

1.4 bibliographic notes & contributions by the author

Parts of this dissertation were already published in peer-reviewed conferences and workshops. In the following, we relate the chapters of this dissertation to these publications and detail the contributions of the author of this thesis.

Chap. 2 and Chap. 3 are based on unpublished material (unless noted) and contributed by Sebastian Biallas. In Chap. 3 the contributions of Sebastian Biallas to the tool Arcade.PLC are listed. The state space for PLC program described in Sect. 2.5.1 is based on ideas first described in [105]. Parsers and translators for Structured Text were contributed by Sebastian Biallas. Other parser were contributed by students under the supervision of Sebastian Biallas. The parser for Statement List was contributed by Andreas Schumacher in his bachelor thesis [108].

Parts of Chap. 4 were first published in [16], on which the reasoning in this chapter is based on. The implementation, formalization and evaluation was performed by Sebastian Biallas. Ideas of Sect. 4.5 were first sketched in [18]. Parts of the case study were first published in [20], [16], [24] and [18]. Additional results were published in [19].

Chapter 5 is based on ideas described in the master thesis of Micro Giacobbe [59], which was written under the supervision of Sebastian Biallas. The results of the thesis were first published in [22], on which Chap. 5 is based on.

Chapter 6 is based on ideas described in the master thesis of Alexander Braining [34], which was written under the supervision of Sebastian Biallas and Volker Kamin. Originally, the approach was implemented for formalizing microcontroller specifications. Later, the approach was geared towards PLC verification. The results of an industrial case study were published in [23]. Sebastian Biallas contributed to the formalization of safety automata and their PLC specific interpretation.

The ideas of Chap. 7 are based on the master thesis by Nico Friedrich [56] and were derived under the supervision of Sebastian Biallas. Parts of Sect. 7.3 were first published in [21]. The reasoning in this section follows closely this publication.

The static analysis approach described in Chap. 8 was first published in [28] and [115]. The value-set analysis to compute summaries described in Sect. 8.7 was first published in [27], [25] and [26]. The analysis of `RETAIN` variables was first published in [68]. The techniques were implemented under the supervision of Sebastian Biallas.

The modular abstraction described in Chap. 9 was first published in [13], which is based on the bachelor thesis of Dimitri Bohlender [32]. The slicing techniques described in this chapter were contributed by Hendrik Simon. The rest of this chapter was contributed by Sebastian Biallas.

During the work on this thesis, the author also published in the field of microcontroller binary code verification [14], microcontroller C code verification [11], pointer analysis of C code [29], abstract interpretation [15], cloud-based soft PLC services [61, 60], and automatic test case generation for PLC code [112, 30]. These works are independent of this thesis.

2

FORMAL VERIFICATION OF PLC CODE

This chapter gives an introduction to PLCs, their modes of operations, and the ways they are programmed. Then, a formal model is derived which reflect their behavior. This formal model is the basis on which we apply formal methods in the later chapters.

2.1 A BRIEF HISTORY OF PROGRAMMABLE LOGIC CONTROLLERS

Initially, hardwired logic was used to implement the controller logic for electrical control devices. Using electro-mechanical devices such as mechanical switches, cam timers as sequencers, relays as controllable switches, coils to store values, it was possible to automate moderately complex control tasks. To exemplify, a simple task such as a logical `AND` between two switches could be implemented by connecting the two switches in series. Similarly, a logical `OR` could be implemented by connecting the switches in parallel. This method had certain advantages such as that a safety function (e. g., two switches have to be pressed to activate a motor) can be implemented in an obviously correct way while ensuring that—physically—the motor is not connected to power if one of the switches is not activated. The disadvantages are that with this method it is harder to implement more complex tasks, and it is very hard to reconfigure, update, or augment the system.

With the invention of the programmable computer, it was possible to store a modifiable/configurable program on the controller (the PLC) that now takes over the control task. This makes it possible to produce universal controllers that can be configured and reprogrammed depending on their task. On the other hand, the more complex nature of computer programs makes reasoning about their correctness much more difficult.

2.2 STATUS QUO

Currently, PLCs are ubiquitously used in the automation and process domain. They are standardized in the international norm IEC 61131 [71]. This, in principle, allows for developing PLC programs independently of their target controllers and simplifies exchange and sharing of functional units between different controllers.

We details the most important aspects of the programming paradigms and the general organization of PLC software in the following.

2.2.1 *Program Organization Units*

PLC programs are usually composed of different modules called *program organization units* (POUs). Each POU consists of an interface providing input and output parameters and an implementation.

Function Blocks

There are three kinds of POUs: *Function Blocks* (FBs) are POUs that also maintain an internal state. To use them, one creates an instance of the FB in the form of a new variable. This variable (the FB instance) can be called, passing values to the input parameters to invoke the FB implementation. The implementation can then access the internal state and the inputs, and so compute new output values, which can be accessed by the caller.

Functions

Functions, on the other hand, only have input and output parameters but no internal state. They can, however, have an explicit return value. Since there is no internal state, it is not possible to instantiate functions. They are called directly, passing new input parameters. It is possible to call a function as part of another expression making use of the return value. It it not allowed to have recursive calls of functions or function blocks.

Programs

Finally, *Programs* are POUs similar to FBs. While FBs only have memory variables as input and output parameters that can be accessed by other POUs, programs can additionally have hardware inputs and outputs. These inputs and outputs are memory mapped or directly connected to hardware ports. An instance of a program runs the control logic as described in the next section.

2.2.2 *Modes of Operation*

Cyclic scanning mode

The most common operation is the so-called *cyclic scanning mode*. This mode comprises three phases which are executed repeatedly as shown in Fig. 3. In the first phase, the input connectors, which are typically connected to sensors, are read and the current values are stored in the input variables of the PLC program. These variables are now fixed for the rest of the cycle. In the second phase, the main POU is called. This program can now compute new values for the output variables based on the current input variables and the internal memory. In the third phase, the final values of the output variables are set to the output connectors, which are usually connected to actuators[1]. These steps are then repeated cyclically to control a process or a machine. Depending on the required sampling rate, real-time requirements, and controlled process, different hardware is available to guarantee

1 After this step, internal tests and internal routines of the firmware of the PLC can be run. This phase is not subject of this work.

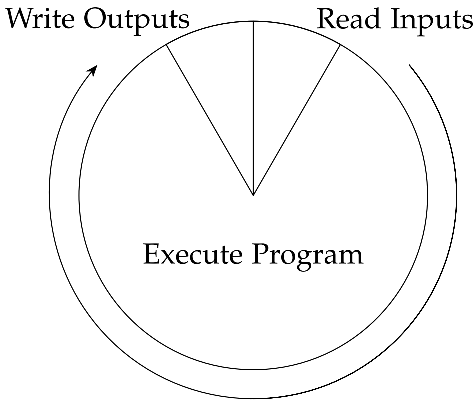


Figure 3: Schematic of the cyclic scanning mode

different cycle times. Typical PLCs have a cycle time of 100 Hz to 1000 Hz, but special-purpose PLCs can operate at up to 1 000 000 Hz.

In the cyclic scanning mode, the program continuously polls for all relevant data from input signals. Additional to this mode, some PLCs also provide *interrupts* that interrupt the current execution of the program for an interrupt handler. The usage of such interrupts is vendor-specific and not handled in this work. *Interrupts*

Observe that in the cyclic scanning mode, each cycle is immediately started after the previous cycle is completed. This entails that the cycle times can be different, depending on the time the program execution takes (if, e. g., the program takes different execution paths depending on the inputs). To cancel computations of a program that is stuck in a loop or just take too long, a watchdog can be used. If the user wants to ensure constant cycle times, the *periodic scanning mode* can be used. It will call the program periodically at fixed points in time. If the program has not finished its computation, a runtime error will be issued. Again, there are vendor specific extensions[2]. In the following, we will only consider programs operating in the cyclic scanning mode according to IEC 61131. *Periodic Scanning*

2.2.3 *Programming Languages*

The functionality of a POU can be implemented in different languages. These languages follow different programming paradigms: Some resemble the electrotechnical background, some reflect an automata based view on a process and others represent typical low- or high-level textual programming languages. Five languages are defined in the standard [71, Part 3]:

2 CoDeSys, e. g., swaps the meaning of periodic scanning mode and cyclic scanning mode. Other vendors call the cyclic scanning mode *continuous scanning mode*.

- *Ladder Diagram* (sometimes called Ladder Logic) is a graphical language that resembles wiring diagrams. Using coils and switches arranged in rungs (hence the name Ladder Diagram), these diagrams follow closely the hard-wired logic of relay circuits.

- *Function Block Diagram* (FBD) is a graphical language that resembles circuit diagrams. An example is shown in Fig. 1 on p. 2. An extension of FBDs, which is used in the process control industry, is called *Continuous Function Chart* (CFC).

- *Sequential Function Chart* (SFC) is a graphical language that allows to specify the sequence of different tasks. They can be run in parallel or in sequence and can be synchronized by different events. The general principles of SFCs are inspired by GRAFCET [72] and Petri nets.

- *Instruction List* (IL) is a textual language resembling accumulator based machine code. An example for an IL program is shown in Fig. 8 on p. 40. A Siemens-specific dialect of IL is called *Statement List*.

- *Structured Text* (ST) is a textual high-level language similar to Pascal. An example is shown in Fig. 12 on p. 60. A Siemens-specific dialect of ST is called *Structured Control Language*.

2.2.4 *Variables, Data Types, Lifetime and Scope*

Each POU can define variables for its interface, for temporary variables used internally, and—in case of a program or a function block—to maintain an internal state. The values of latter variables are then retained between different invocations. The standard [71, Part 3] defines different data types for variables. In the following, we present the different data types defined by the standard and some prominent vendor-specific extensions. For each category of types, we briefly mention whether we support these types for the purpose of our verification work: *unsupported* means that we cannot handle these types, *ignored* means that we support declaring and using these types but ignore all operations. If the ignored types are tested or converted into supported types, we over-approximate the behavior and assume that each value might occur.

Table 1 shows the different integer and floating point types that can be used for variables. The difference between the bitwise and normal integers is that bitwise integers can be used for bitwise logical operations. It is possible to convert between all data types using a *xx*_`TO`_*yy* function, where *xx* is the source and *yy* is the destination data type. It is possible to subtype these scalar types to only support a limited range. It is also possible to define enumerations as `ENUM` types that then offer a list of named constants. There are additional *binary coded decimal* (BCD) types, which are unsupported in this work.

Data type	Size	Range	Remark
`BOOL`	1 bit	`true/false`	Bitwise
`USINT`	8 bit	$0 \ldots 2^8 - 1$	
`BYTE`	8 bit	$0 \ldots 2^8 - 1$	Bitwise
`UINT`	16 bit	$0 \ldots 2^{16} - 1$	
`WORD`	16 bit	$0 \ldots 2^{16} - 1$	Bitwise
`UDINT`	32 bit	$0 \ldots 2^{32} - 1$	
`DWORD`	32 bit	$0 \ldots 2^{32} - 1$	Bitwise
`SINT`	8 bit	$-2^7 \ldots 2^7 - 1$	
`INT`	16 bit	$-2^{15} \ldots 2^{15} - 1$	
`DINT`	32 bit	$-2^{31} \ldots 2^{31} - 1$	
`REAL`	32 bit	IEEE 754 single[3]	Floating-Point
`LREAL`	64 bit	IEEE 754 double	Floating-Point

Table 1: Scalar Types

Strings can be stored in different string types that support different character encodings and strings of bounded and unbounded length. We ignore the usage of strings for the purpose of this work.

Types can also represent aggregates of other types. A `STRUCT` combines different data types into an aggregation, where each member can be accessed by its name. The most recent standard allows for overlapping data structures in the form of unions, which we do not support. Aggregations of the same data type can be stored in an `ARRAY`. Their members can only be accessed using an index expression. Optionally, the index can be multi-dimensional. Accessing non-existing array elements using an index which is out of bounds can yield to runtime errors. Hence, we added an analysis to detect these errors (cp. Sect. 8.5). In some vendor-specific extensions, the distinction between structs and arrays is blurred, cp. Sect. 8.6.2.

The most recent standard IEC 61131-2013 defines *references* as a variable type. These references contain the address of another variable and thus allow for indirect access. In vendor-specific extensions, *pointers* are used with similar semantics. In our framework, we handle references and pointers under a unified concept (cp. Sect. 3.4.4).

Function Block Instance

Scope of Variables

A function block is also a variable type. A variable of function block type is called an *instance* of a function block. Each instance gets its own set of variables defined in the function block type. These variables have a certain *lifetime* and *scope* depending on whether they retain the value between different invocations of the function block and whether they can be accessed from the outside scope. The most important variable declarations and their lifetime and scope are shown in Tab. 2.

3 According to the most recent IEC 61131-2013 standard, the ranges of the `REAL` and `LREAL` data types are defined according to the IEC 60559 (IEEE 754) *single* and *double* types.

Declaration	Lifetime	Scope
`VAR_INPUT`	Set to new value each invocation	Accessible from outside
`VAR_OUTPUT`	Retains value for next invocation	Accessible from outside
`VAR_TEMP`	Not stored	Only internal
`VAR`	Retains value for next invocation	Only internal
`VAR_GLOBAL`	Retains value for next invocation	Accessible everywhere

Table 2: Lifetime and scope of different variable types

Retain and Persistent Variables

Additionally, variables can be marked as `RETAIN` and `PERSISTENT`. These modifiers control whether the value of a variable is retained between a reset of the PLC. To ensure the storage even between hard resets (power switch off), the values might be stored in, e. g., permanent memory. While the exact semantics of retain and persistent variables is vendor-specific, the concept of persistent variables can be a source of errors if the programmer forgot to mark some variables as retained, since this can result in inconsistent configurations after a restart. In Sect. 8.4.1, we will analyze how we can detect such situations.

2.2.5 *General Organization*

A typical PLC program is modularly composed of different POUs. Each POU can include other POUs: It might call other functions or instantiate other function blocks as local variables. These POUs can then, in general, be implemented in different languages. A programmer might, e. g., implement the high-level control logic in SFC and then implement the low-level FBs in ST. Recursive calls of POUs are not allowed, i. e., recursive function or function block calls are not allowed and the nesting of POUs must be finite. This simplifies the analysis of PLC programs, since all local variables have a unique address.

In Fig. 4, a typical controller configuration is depicted. Different tasks having different cycle times can be run on one controller. Each task is represented by one program, which can instantiate function blocks and has locale variables representing the current state. Additionally, global variables (i. e., variables that all programs can access) can be used to represent shared memory between the tasks.

The standard already defines functions and function blocks for the most common operations. This includes function blocks for flip-flops, edge detection, timers, counters, mathematical functions and function for string manipulation.

Safety-Critical Applications

When developing safety-critical applications, the programmer is sometimes restricted to certain languages and constructs. Typically, a certified development environment is needed that only allow the connection of certified function blocks using an FBD. An example for such a library is given in Sect. 2.3. An example for a safety application developed using this library is shown in Fig. 1 on p. 2.

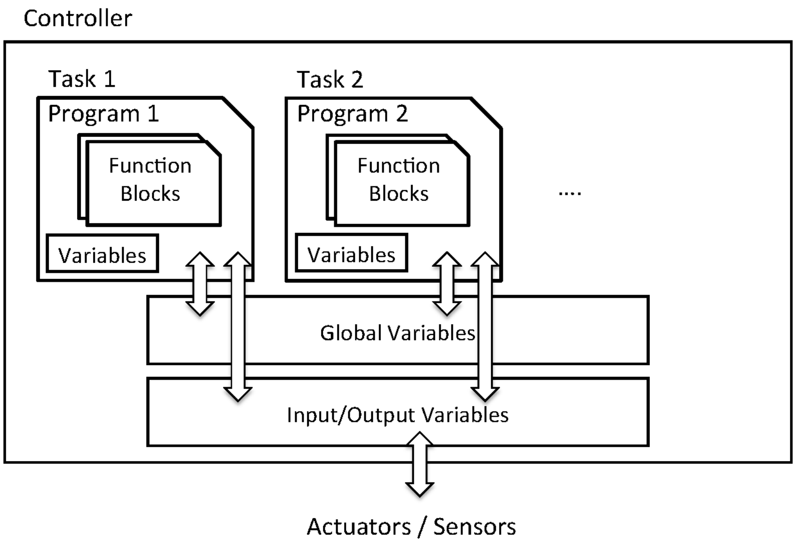


Figure 4: Typical Organization of a PLC

2.2.6 *Timers*

PLC programs can make use of *timers* to control or measure the duration of events. Such timers are used to, e. g., invoke (or cancel) processes after a certain time, to detect timeouts, or to detect signal equivalency with a certain discrepancy time. There are two conceptually different kinds of timers available. The first kind provides a current time in, e. g., milliseconds. An example of such a timer would be the `TIME()` function available in CoDeSys environments, which returns the current time. When invoking this function multiple times in one cycle, the function might return different values because a certain amount of (actual) time has passed.

By way of contrast, PLCs also expose a second kind of timer FBs that provide a "frozen" state of the timer during the cycle. That is, at the beginning of the cycle a memory image of the timer will be provided, which provides a consistent view of the timer: This behavior is similar to the input variables of the program, which are also fixed at the beginning of the cycle and do not change during the cycle. The standard defines the three timer FBs `TON`, `TOF`, and `TP` that behave this way.

2.2.7 *Function Block Calls*

Passing parameter values to function blocks or functions can be performed using two different methods. First, they can be passed directly when calling a function block. To illustrate, let `functionblock` be a function block with input `input1`, `input2` and output `output1`. A direct call can be performed as follows:

```
functionblock(input1 := 1, intput2 := a, output1 => result);
```

This would pass the value 1 to `intput1`, the value of *a* to `intput2`, and copy the value of `output1` into `result` after the call. If copying of output parameter is not necessary, the parameter values can also be passed without explicitly naming the input variables (in this case, the order in which the values are given is important):

```
functionblock(1, a);
```

Secondly, the function block can be called without specifying any parameters. In this case, the current value of the input variables is used. To change or read the value, these variables are accessible from the caller's scope. Passing the parameter as per the example given above would look as follows:

```
functionblock.input1 := 1;
functionblock.input2 := a;
functionblock();
result := functionblock.output;
```

In practice, a mixture of these styles is used. In Sect. 8.3, we detail a technique to reduce the visibility of these variables for a static analysis framework.

2.2.8 *Standard & Vendor-Specific Extensions*

Although PLC programming languages are standardized [71, Part 3], actual implementations differ slightly between vendors or even within the same vendor. During the course of this dissertation, especially syntactic differences of the ST dialect between different vendors had to be accounted for. As we will see in Sect. 3.4.3, we handle this using a specialized grammar.

Short-Circuit Evaluation

The standard does not define whether Boolean expressions are evaluated using *short-circuit evaluation*, which would mean that the evaluation of an expression is not stopped once the outcome is known. To exemplify, when evaluating an expression such as `A AND B` and `A` is known to be false, one does not have to evaluate `B`. All vendors we checked always evaluate all sub-expressions, without short-circuit evaluation. This can be counter-intuitive, especially for programmers who are used to short-circuit evaluation. As an example, consider the following code fragment:

```
IF B<>0 AND A/B > 1 THEN
   (* .. *)
END_IF;
```

The intention of the programmer was that the division `A/B` is only performed if `B` is not zero. Yet, this division is performed since all sub-expressions are always evaluated. We also evaluate all sub-expressions, so we can detect the error in this case. CoDeSys additionally implements an extension using the keywords `AND_THEN` and `OR_ELSE` to force short-circuit evaluation, which we do not support.

The standard does not always define a clear result for certain operations, or even forbids them. Instances of such undefined or disallowed constructs are overflow of data types, division by zero, and array accesses outside the array bounds. Consequently, this often indicates an error in the program. Depending on the vendor and language dialect, these constructs can either trigger a runtime error or even unwanted side effects. Although these situations are not always fatal or might have defined results for some vendors, we catch them during our static analysis.

2.3 PLCOPEN

PLCopen is a consortium that works on the standardization and harmonization in the field of industrial automation. They develop new standards and improve existing ones such as the IEC 61131. In particular, they define certain standard FBs for the application in various domains, e. g., for motion control or safety. These FBs are defined in a vendor-neutral way and are then typically implemented by library authors for different PLCs. The idea is to have a set of typical FBs that are needed in many applications and thus to reduce the costs of reimplementing and validating these blocks. The FBs are not directly implemented by PLCopen, but only described using different means [94]:

- A textual specification is provided for selected important properties of each block.
- The behavior is exemplified using digital timing diagrams.
- A semi-formal specification of the complete behavior is given as a so-called *state diagram*. These state diagrams comprise states, which specify the output behavior, and transitions between states, which are triggered by certain input values or timers. An example for a state diagram is shown in Fig. 18 on p. 83. They also inspired our automata based specification in Chap. 6.

Safety Function Block Library

We implemented our own version of the PLCopen safety function block (SFB) library in Structured Text. We use PLCopen SFBs from our own implementation, from another group [114], and for an industrial implementation in our case studies (cp. Sect. 4.6, Sect. 6.4, and Sect. 8.6.1).

PLCopen XML

AutomationML

Additionally, PLCopen defines the PLCopen XML file format, as a standardized exchange format between different PLC development environments and tools [71, Part 10]. It became part of the AutomationML file format (Automation Markup Language) as standardized in [73]. This XML format offers topological and geometrical information about plants and machines, connection information between sensors, actuators and controllers as well as the actual control logic, which is then encoded in PLCopen XML. We implemented a parser for the PLCopen XML format that extracts the control logic of PLC programs into our verification tool (cp. 3.4.1).

2.4 FORMAL VERIFICATION USING MODEL CHECKING

Originally, model checking was used to verify concurrent processes [39, 4, 42]. We use model checking to verify the correct input/output behavior of PLC programs. In this section, we formally introduce model checking and then describe the model that we are verifying.

2.4.1 *Kripke Structures*

We model the PLC behavior in a labeled transition system. Therefore, let P be a set of atomic propositions which will act as the labels. In the context of PLCs, these are propositions over the inputs and outputs of the program or the internal variables of the program.

Labeled Transition System

Definition 2.1: A *labeled transition system* is a tuple $\langle S, R, L\rangle$ with

- a finite set S of states
- a transition relation $R \subseteq S \times S$
- a labeling function $L : S \to 2^{AP}, \quad L(s) = \{f \in AP \mid s \models f\}$

Kripke Structure

Such a transition system naturally describes the behavior of a discrete event system such as a PLC. A transition system $\mathcal{M} = \langle S, R, L\rangle$ with an initial state $s_0 \in S$ is called a *Kripke structure*. The test, whether a Kripke structure is a *model* of a logical formula φ, denoted:

$$(\mathcal{M}, s_0) \models \varphi,$$

is called *model checking*.

Typical logics that have Kripke structures as models are LTL (*linear time logic*), CTL (*computation tree logic*) and CTL* (an extension of CTL) [53]. These logics are able to express properties about paths (i. e. temporal behavior[4]) and branches (i. e. non-deterministic behavior).

2.4.2 CTL *Formulae*

For a set P of atomic propositions, a CTL formula is inductively defined as follows:

- *TT* and *FF* are state formulae.
- Each $p \in P$ is a state formula.
- For state formulae φ, ψ the formulae $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\varphi \Rightarrow \psi$ are state formulae as well.

---

4 Here, the term *temporal* refers to in which order things happen but not to physical time.

- For state formulae φ, ψ the formulae $G\varphi$, $F\varphi$, $X\varphi$ and $\varphi U \psi$ are path formulae.
- For a path formula φ the formulae $A\varphi$ and $E\varphi$ are state formulae.

Each state formula defined this way is a CTL formula. For a transition system $\langle S, R, L \rangle$ the semantics of a state formula for a state s is defined as follows:

- *TT* and *FF* denote *true* and *false*.
- The atomic propositions and the operators $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ are used as in propositional logic for the labelling $L(s)$.
- For a path formula φ the all-quantified state formula $A\varphi$ is fulfilled if all paths starting in s fulfill φ.
- For a path formula φ the existential-quantified state formula $E\varphi$ is fulfilled if at least one path starting in s fulfills φ.

Now let $\pi = \langle s_0, s_1, s_2, \ldots \rangle$ be a path. The semantics of a path formula for state formulae φ, ψ w. r. t. π is defined as follows:

- $G\varphi$ (*globally*) is fulfilled if all states on the path fulfill φ, i. e., $\forall i : s_i \models \varphi$.
- $F\varphi$ (*finally*) is fulfilled if at least one state on the path fulfills φ, i. e., $\exists i : s_i \models \varphi$.
- $X\varphi$ (*next*) is fulfilled if the next state on the path fulfills φ, i. e. $s_1 \models \varphi$.
- $\varphi U \psi$ (*until*) is fulfilled if there is a state on the path that fulfills ψ and all previous states fulfill φ, i. e., $\exists i : s_i \models \psi \wedge \forall j < i : s_j \models \varphi$.

CTL is characterized by the fact that branch operator (A, E) are always combined with path operator (G, F, X, ..U..). In CTL* this restriction is lifted. In this case, however, it is no longer possible to efficiently check formulae [4]. By ∀CTL, we denote the all-quantified fragment of CTL [53]. CTL* ∀CTL

2.4.3 *Counterexamples and Witnesses*

If an all-quantified formula φ is violated then there exists a trace in the state space that violates φ. Such a trace is called *counterexample*. For a safety-property, a counterexample is a finite path ending in a state violating this property. For a guarantee-property, a counterexample is an infinite path (i. e., a loop) that does not reach a required property. For existential-quantified properties, on the other hand, we can generate a witness that proves the property.

For a user, counterexamples are very helpful in understanding why a formula is violated [39]. Counterexamples also play a central role in Chap. 4, where they are used to refine an abstraction. In Chap. 7, we will present techniques to automatically locate the problematic steps, i. e., the steps that are most likely responsible for a violation and thus represent the bug in the program. An example for how a counterexample for PLC program looks is shown in Fig. 20 on p. 89.

2.5 MODEL CHECKING PLC PROGRAMS

In this section we define a formal model for PLC programs, which can be used to verify certain properties. These properties can be simple invariants, such as *if an input is set, a certain output has to be set as well* but can also specify the order of events, such as *if an input is set, then an output must be* 0 *until another input is set*. Since we also want to verify functions or functions blocks, we generalize this model to all kinds of POUs.

2.5.1 *Concrete Model*

A key aspect of our formal model is that we want to capture only the observable behavior of the PLC program. That is, we are interested in a particular stimulus and the response of the PLC program. The stimulus corresponds to values of the input variables of program (and some extra information, e. g., which timers are about to fire this cycle). The response then corresponds to the values at the outputs at the end of the cycle. Everything that happens during the execution of the cycle is not observable and thus should not be subject to the verification. Note that exposing internal states might even cause spurious errors: Suppose two outputs, `output1` and `output2`, should both contain the same value and they are both 0 at the beginning of the cycle. Suppose further, that the following ST code is used to set them both to 1 during the program:

Observable Behavior

```
23 // [..]
24 output1 := 1;
25 output2 := 1;
```

Then, after the execution of line 24 an error would be signaled, since `output1` is 1 but `output2` is still 0. For an actual PLC program, the effect of the two assignments would be visible only at the end of the cycle and thus provide a consistent (and correct) result. It is hence necessary to verify properties only at the end of the cycle, which is what we want to reflect in our model.

In principle, our model is a transition system between PLC states. A state can be seen as a memory dump of the PLC variables, or, more formally, an assignment function of the PLC variables:

Variables

Definition 2.2: We denote by $\textsc{Var}$ the set of all variables of the program and by $\mathcal{D}$ the domain of $\textsc{Var}$. We partition $\textsc{Var}$ into $\textsc{Var} = \textsc{Var}_M \dot{\cup} \textsc{Var}_I$, where $\textsc{Var}_M$ represents all variables that retain their value for the next cycle, whereas $\textsc{Var}_I$ represents the input variables that are assigned a new value each cycle (cp. Tab. 2 on p. 14). Further, we call an assignment $\textsc{Var}_I \to \mathcal{D}$ to the inputs an *input configuration*. We use the set $\textsc{Var}_T$ to refer to temporal variables, i. e., local variables that do not retain their value for the next cycle. Observe that $\textsc{Var}_T \cap \textsc{Var} = \{\}$, since temporal variables are not part of the model.

Input Configuration

Definition 2.3: Let $\mathcal{D}$ be the domain of all variables VAR occurring in the program. Then a *memory state* is an assignment $s : \text{VAR} \to \mathcal{D}$ reflecting the configuration of the PLC after the execution of a cycle. If the context is unambiguous, we will call memory states just states. Often, we will write states explicitly as a tuple of assignments such as $\langle out = 0, var = 0, \dots \rangle$. There is one state, called the *initial state*, which is characterized by all variables set to their initial values. This is the state when the PLC is switched on and usually denoted by s_0.

Memory State

Definition 2.4: The *(explicit) model* $(\mathcal{M}, s_0)$ of a PLC program is a transition system $\mathcal{M}$ between states $s \in S$, where s_0 is the initial state. The transition relation $R \subseteq S \times S$ is characterized as follows: State $(s, s') \in R$ iff s' is reachable from s after one PLC cycle. In other words, if the PLC is configured according to s and there is an input configuration such that after the invocation of the program, the PLC is configured according to s', then (and only then) $(s, s') \in R$. In this case, we call s' a *successor* of s.

Explicit Model

Successors of a State

We can build the explicit model from a program automatically: Starting from s_0, we enumerate all possible input configurations and execute the program, discovering all successors s_0. Then, for each successor, we repeat this process, discovering the next level of successors. Since the size of the memory states is finite, this process will eventually terminate with the complete state space of the program.

Note that our model abstracts time. Each transition in the Kripke structure represents one cycle of the program but we have no accurate timing information. We therefore, over-approximate timer function blocks (cp. Sect. 2.2.6): Once a timer is started, we assume that it can fire in each cycle. Once it has fired, it can no longer fire until it has been restarted. Hence, we can only prove properties regarding the order of certain events but not regarding how long certain operations take. In practice, however, the Boolean `Q` outputs of timer function blocks can be used in formulae to specify that certain property only happen if or after a timer has fired.

Time

2.5.2 *Abstract Model for PLC Programs*

The construction of the concrete state space introduced in the previous section is very susceptible to state explosion, since state spaces grow exponentially in the number of inputs [16]. To make this approach feasible, we now turn to building abstract state spaces that combine sets of concrete states into macro states.

Formally, this abstraction can be seen as a partition of the state space. This partition induces an equivalence relation $\sim$ of states. We write $s \sim t$ if the states s and t lie in the same equivalence class. This equivalence class represents the macro state

Equivalence Relation

$$s/\!\sim \quad := \quad \left\{ s' \in S \mid s' \sim s \right\}, \tag{1}$$

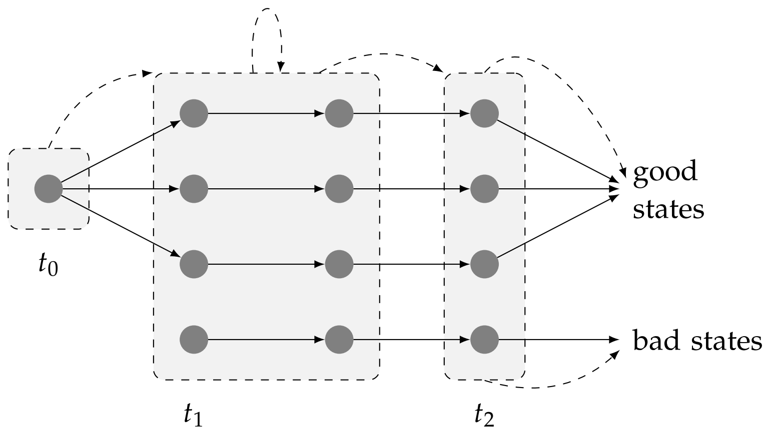


Figure 5: Kripke structure with abstraction.

which summarizes all states equivalent to s. Using an equivalence relation $\sim$ and a concrete Kripke structure $(\mathcal{M}, s_0)$ we obtain an abstracted Kripke structure $(\mathcal{M}', s'_0)$ with $\mathcal{M}' = (S', R', L')$ defined as follows:

$$\begin{aligned} S' &= \{s/\sim \mid s \in S\} \\ s'_0 &= s_0/\sim \\ R' &\subseteq S' \times S' \\ (s', t') \in R' &\Leftrightarrow \exists (s, t) \in R : s \in s' \wedge t \in t' \\ L'(s') &= \bigcup_{s \in s'} L(s) \end{aligned}$$

Such an abstraction is usually called *existential abstraction* [42]. Figure 5 shows an example of such an abstraction. The concrete states are indicated as solid circles and the three macro states t_0, t_1 and t_2 are indicated by the dashed lines. The abstracted transitions are indicates by the dashed arrows. We hence abstracted 13 states by 3 macro states. Observe that in the concrete state space we cannot reach a bad state from t_0. The abstracted state space, however, admits a path $t_0 \to t_1 \to t_2$ that reaches the bad states. This a spurious counterexample, which occurs since our abstraction allows for more behavior than the concrete state space.

Spurious Counterexamples

Refinement

To suppress spurious counterexamples, we have to *refine* an abstraction, i. e., treat states differently that were summarized in the same equivalence class before. In this case, it help to split the states at the bottom in Fig. 5 into another macro state. The refined state space is shown in Fig. 6. Observe the new equivalence classes t'_2 and t'_5 that we introduced. These classes summarize the states that yield to the bad states. Now, that they lie in the our equivalence class, the abstracted state space is safe: The bad states are not reachable from the initial state.

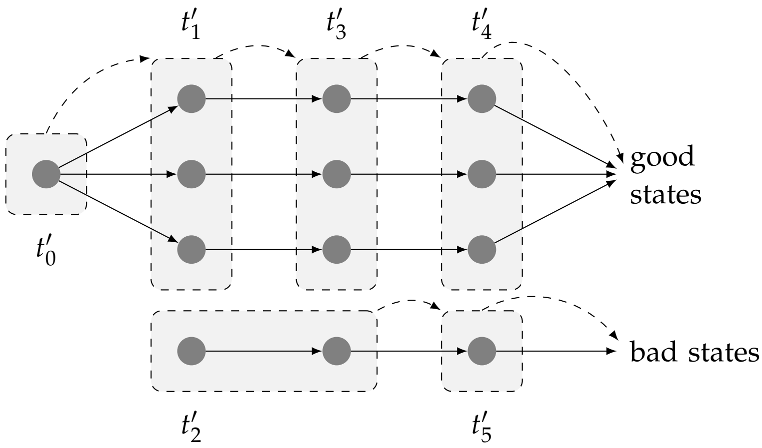


Figure 6: Refinement of the abstraction shown in Fig. 5

The techniques we presented here, i. e., using a spurious counterexample as an indicator to refine a given abstraction give rise to the counterexample guided refinement schemes [40]. We use such a scheme in Chap. 4 to automatically generate a refined abstraction of PLC programs for model checking.

3

IMPLEMENTATION

During the course of this dissertation, we created the tool ARCADE.PLC. In this chapter, we describe the history of ARCADE.PLC and how the model checker and the static analysis is organized. We then describe the abstract simulator used for efficient creating of abstract state spaces and the intermediate representation used to offer a canonical interface for the verification of programs written in different languages.

3.1 ARCADE.PLC

History

ARCADE.PLC originated as an offspring of the [MC]SQUARE model checker [103, 106]. The development of [MC]SQUARE started in 2004 as a model checker for microcontroller code. [MC]SQUARE directly works on machine code and thus contains simulators and hardware models for various microcontrollers such as ATMEGA ATmega16/256, C51 and Renesas R8C [92, 100, 101, 99]. To make the adoption to new microcontrollers easier, it also contains a generator to automatically generate specific analyses from machine descriptions [67, 66]. In 2009, the possibility to verify PLC programs written in Instruction List was added [105].

To better reflect the different applications and areas of expertise, [MC]SQUARE was renamed to ARCADE[1] and split in the sub-projects ARCADE.μC for microcontroller verification and ARCADE.PLC for PLC verification. Both projects share an Eclipse-based user interface built on the Rich Client Platform. For ARCADE.PLC, it allows for inspecting programs written in Instruction List (IEC and Siemens), Structured Text, and Function Blocks Diagram. Possible errors and warnings can directly be highlighted in the source code.

Graphical Front-End

Additionally, we developed a command line interface that allow for running the model checker or the static analysis from batch processes or other tools. This is especially useful for a continuous integration environment where each new version is automatically tested. For each new revision, one can, e. g., automatically check whether previous requirements are still fulfilled, or check whether the static analysis detects new warnings. Finally, we also developed a server-based interface for the static analysis of ARCADE.PLC. This server is used to implement a web-based

Command Line Interface

1 ARCADE stands for *Aachen Rigorous Code Analysis and Debugging Environment*.

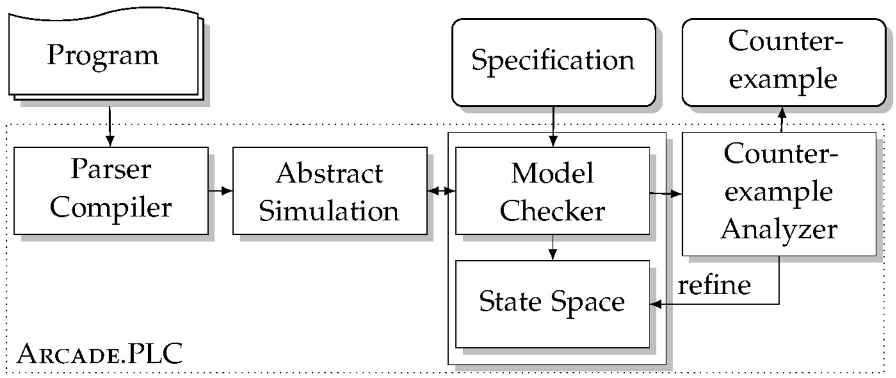


Figure 7: Model checking process with ARCADE.PLC [18]

Web-Based Front-End front-end to showcase the static analysis capabilities. Since the server also produces machine readable results, it can easily integrated in other development platforms or work flows without the need for a local installation of the ARCADE.PLC tool.

Libraries ARCADE.PLC ships with three PLC libraries that can readily be used in all programs: Our standard library covers most function and function block defined by IEC. We also offer a PLCOPEN safety function block library in a `PLCOPEN` namespace. This library was written during the course of this dissertation. We use this library in various case studies (cp. Sect. 4.6 and Sect. 5.6). This library can be selected if no vendor-specific PLCOPEN library is provided. Finally, we also provide helper functions in the `ARCADE` namespace that, e. g., allow for checking user-provided invariants in the form of *assertions* (cp. Sect. 8.5, p. 121).

3.2 ORGANIZATION

The overall organization of the model checking process of ARCADE.PLC is depicted in Fig. 7. It contains the following components:

- Parser and compiler for different PLC languages. These are described in Sect. 3.4.1.
- An abstract interpreter that can be parametrized. It is used for building the state space for the model checker as well as the static analysis and described in Sect. 3.3.
- A model checker for verifying CTL and automata-based specifications.
- Automatic refinements for the model checker, described in Chap. 4 and 5.

Additionally, ARCADE.PLC contains a static analysis, which we describe in Chap. 8.

3.3 GENERIC SIMULATOR AND ABSTRACT DOMAINS

The simulator of ARCADE.PLC is written in a way that its operations can be parameterized: On the one hand, they can operate on concrete values yielding a PLC simulator. This simulator is powerful enough to run a soft-PLC[2] in industrial contexts [61, 60].

On the other hand, the simulator of ARCADE.PLC can be configured to operate on abstract domains. Abstract domains summarize a sets of concrete values into an abstract value. For each abstract domain, all operations provide a sound over-approximation of the concrete program semantics: Let $\mathcal{C}$ be a concrete domain (e. g., $\mathbb{Z}$) and $\mathcal{A}$ an abstract domain (e. g., $\mathcal{A} \subseteq \wp(\mathcal{C})$, where $\wp(S)$ denotes the power-set of S). A *concretization function* $\gamma : \mathcal{A} \to \wp(\mathcal{C})$ maps an abstract value $a \in \mathcal{A}$ to the set of concrete values it summarizes. An *abstraction function* $\alpha : \wp(\mathcal{C}) \to \mathcal{A}$ maps a set $C \subseteq \mathcal{C}$ to its abstract counterpart $\alpha(C) \in \mathcal{A}$ such that

Concretization Function

Abstraction Function

$$C \subseteq \gamma(\alpha(C)). \tag{1}$$

This ensures that no value is "lost" during abstraction. To abstract the program semantics, let $f : \mathcal{C} \to \mathcal{C}$ be an operation of the program. To abstract the behavior soundly, we need an abstract transformer $f^{\#} : \mathcal{A} \to \mathcal{A}$ of the operation such that

$$f(\gamma(a)) \subseteq \gamma(f^{\#}(a)) \text{ for all } a \in \mathcal{A}. \tag{2}$$

Intuitively, this means that all program behavior is tracked when abstracting the program semantics. We allow, at most, for more behavior of the program.

In the following, we describe the abstract domains that have been implemented for abstract simulation. These domains can be used for model checking as well as the static analysis. All domains share the same interface such that the simulation can be performed on any of these domains. The key component, however, is a reduced product of these domains, which we introduce in Sect. 3.3.5. Using this product domain, all operations can be performed on all domains in parallel. Domains can then share and propagate information, increasing the overall precision. We first define lattices, which form the background of all domains.

Abstract Domains and Reduced Product

3.3.1 *Lattices*

Definition 3.1: A *lattice* $(L, \sqsubseteq)$ is a partially ordered set (po-set) L w. r. t. $\sqsubseteq$, in which every two elements e_1, e_2 have a unique supremum $e_1 \sqcup e_2$ (called the join operator) and a unique infimum $e_1 \sqcap e_2$ (called the meet operator).

Lattice

In our case, we assume that each lattice has a maximum element $\top$ and a minimum element $\bot$. The join operator can intuitively be seen as a union operator that merges information from different points. The meet operator can be seen as an intersection between different objects. A lattice can hence collect the semantics of a

2 A soft-PLC is an industrial PC that runs PLC programs usually using a real-time OS.

program during abstract simulation. The $\top$ element can then be seen as *everything is possible* and the $\perp$ element can be seen as *nothing is possible*.

3.3.2 *Intervals*

Intervals are the most common abstract domain [44]. We abstract a set S of values as $\alpha(S) = [min(S), max(S)]$ and store the interval as a tuple. There are special $\top$ and $\perp$ elements for a full and empty set, respectively. The meet operator is implemented as an interval intersection, whereas the join operator selects the minimum and maximum from both operands. The abstract transformers for linear arithmetic operations can then be defined as operations on the interval bounds, providing a sound abstraction due to the linearity. Many non-linear operations such as bit-wise operations cannot be modeled precisely using intervals and thus incur an over-approximation. We hence extend the interval bounds up to the range of the variable type in these cases.

Example Consider $[5, 7] + [2, 2] = [7, 9]$ which is exact (all values of the result are actually feasible). In contrast, we have $[5, 7] * [2, 2] = [10, 14]$, which contains the values 11 and 13 not being multiples of 2.

3.3.3 *Bitsets*

Bitsets are used for abstracting Boolean logic and bitwise operations [16, 28]. They are represented as bit-vectors $\langle b_n, \ldots, b_1, b_0 \rangle$ (for a variable of n bits), where each bit b_i is either 0, 1 or *unknown* (denoted $*$), where *unknown* means that we do not know whether the bit is 0 or 1. All Boolean and bit-wise operations are then modeled on the bit-level using three-valued logic. Other operations are not supported and return $\top$, i. e., all bits unknown. To illustrate, let AND and OR be the Boolean operations. Then

Example

- $\langle 1, 0, 1, * \rangle \ \mathsf{AND} \ \langle *, 1, 1, 0 \rangle = \langle *, 0, 1, 0 \rangle$,
- $\langle 1, 0, 1, * \rangle \ \mathsf{OR} \ \langle *, 1, 1, 0 \rangle = \langle 1, 1, 1, * \rangle$, and
- $\langle 1, 0, 1, * \rangle + \langle 0, 0, 0, 1 \rangle = \perp$.

Hence, arithmetic that involves unknown bits incurs a loss of precision. While the last operation could in theory be summarized as $\langle 1, 1, *, * \rangle$, we abstract arithmetic operations to return an unknown result. We chose this approach for simplicity, since, as we will see, such operations can already be captured by intervals and the reduced product introduced in Sect. 3.3.5.

3.3.4 *Extensions*

We provide two other domains to make intervals more precise. First, to abstract a small number of distinct values precisely, we use *k-sets*. In principle, these are sets

k-Sets

that cannot contain more than k values (indicated by a subscript k in the following). We denote by the special symbol $\{*\}_k$ any set that contains more than k values. This is equivalent to the $\top$ element of this domain, representing all possible values of the variable type.

Let $m_1 = \{5, 7, 18\}_4$, $m_2 = \{12\}_4$, $m_3 = \{12, 13\}_4$ be 4-sets. Then $m_1 \sqcup m_2 = \{5, 7, 12, 18\}_4$, whereas $m_1 \sqcup m_3$ could only be represented by $\{*\}_4$, since it is not representable exactly as a 4-set. *Example*

During the abstract simulation, all operations are performed for each value in the set. This explains the restriction to, at most, k values per set, so as to maintain efficient operation. The force of the k-set domain is that variables that only hold a small number of values during the execution can be represented exactly. This especially includes diagnosis codes, enumeration types and program states. Variables which range over a huge number of values (e. g., sensor values), on the other hand, are abstracted using the $\{*\}_k$ symbol, which makes their handling still efficient yet imprecise. Currently, the value of k can be configured manually, and is 50 by default. In practice, this seems to be a good compromise between precision and speed.

Additionally, we provide interval-sets as an extension of the k-sets. Interval-sets are stored as a set of intervals offer a precise join operator. To illustrate, we have $\{[1, 2]\} \sqcup \{[4, 5]\} = \{[1, 2], [4, 5]\}$ with interval-sets, whereas $[1, 2] \sqcup [4, 5] = [1, 5]$ with intervals. All operations between interval sets have to be performed as a cross product of all interval combinations. Afterwards, the result has to be normalized, i. e., overlapping and adjacent intervals have to be merged. Since this incurs an overhead for each operation and, additionally, so as to avoid an explosion in the number of intervals stored, we use a threshold for the maximum number of intervals, similar to the k-sets. After more than this threshold intervals are stored, intervals are merged. While this loses precision, it still provides a sound over-approximation.

3.3.5 *Reduced Product*

Each domain offers a trade-off between precision on the one hand, efficient operations and in-memory representation on the other hand. To combine them, we provide a (partially) *reduced product* domain [44], which allows for running all abstraction operations in parallel in different domains. This is an approach similar to [98, 35]. By using this combination and performing each operation on each domain, we are able to precisely capture a variety of different program behaviors: Intervals deal with integer arithmetic, bitsets are suitable for Boolean logic and k-sets accumulate small sets of distinct values. Since we also allow information exchange between domains (if an abstract value represents a single value in one domain, this information is propagated to the other domains), the precision is further increased.

3.4 TRANSLATION TO THE INTERMEDIATE REPRESENTATION

To handle the different PLC languages and to handle PLC programs written in a combination of different languages under one framework, we translate all programs into an intermediate representation (IR).

3.4.1 *Parsers*

We allow for loading PLC files from plain text files or from AutomationML [73] files in the PLCopen XML format. These files can contain multiple POUs written in different languages.

Instruction List For Instruction List according to the IEC 61131 standard, we build on a parser from a previous work [105]. In contrast to this work, which directly simulates the IL semantics, we first translate IL into our IR so as to have a canonical platform for all further analyses. IL is an accumulator based machine languages on which all logical and arithmetic operations are performed. We, therefore, introduce an accumulator variable (which has temporary lifetime) to reflect all operations. Figure 9 on p. 43 shows an example of this translation.

Statement List We also implemented a parser for *Statement List* (the Siemens dialect of Instruction List). It is handled similar to Instruction List according to IEC. Here, we have to model different accumulators. The accumulators are represented as 16-bit words, where the upper and lower byte can be accessed separately. To represent the handling of these accumulators efficiently without too much overhead, we model the accumulator word and its byte as separate accumulators. As long as the accumulator is accessed either in a byte-wise or in a word-wise fashion, we require no overhead. If, on the other hand, byte- and word-wise access is intertwined, we have to insert instructions to convert the different accumulator representations into each other. We resolve this during the translation by a series of bit-shift and masking instructions.

Function Block Diagram Function Block Diagrams are read from the PLCopen format. We offer an interactive FBD editor that allows for configuring the order in which FBs are evaluated. The semantics of an FBD is then translated into a series of `Call` instructions in our IR.

Structured Text For Structured Text, we implemented a parser which generates an abstract syntax tree (AST). This AST is then compiled into the IR. While this transformation is straightforward, several practical considerations had to be taken into account, which we detail in the next sections.

3.4.2 *Annotations using Pragmas*

To guide the parser or the analysis, we allow to annotate the source code using *pragmas*. A pragma is an annotation in the source code that is syntactically a comment (i. e., it is ignored by the parser), yet can bear special meaning in some contexts. We use the syntax `{@text...}` for pragmas, since the text between curly braces is usually ignored by other parsers, similar to comments. The most recent IEC standard recommends the curly braces for such constructs. We define the following pragmas:

`{@ARCADE CONTEXT-SENSITIVE:TRUE}`	Analyze instances of the POU in a context-sensitive way, cp. Sect. 8.2.2.
`{@CHECK PRECONDITION condition}`	User-defined check for POUs.
`{@DIALECT dialect}`	Switch between different dialects, see next section.

The `CHECK` pragma allows for defining preconditions for POUs. The argument `condition` can be an arbitrary condition over the variables of the POU and is checked each time an instance of the POU is called. The checks are performed during the static analysis (cp. Sect. 8.5).

3.4.3 *Pragmatic & Practical Considerations*

Vendor-specific Dialects

Although the languages are standardized, many different vendor-specific dialects are used to write industrial PLC programs. We implemented support for ST program written in the dialects IEC 61131 (version 1993, 2003 and parts of 2013), CoDeSys, and Siemens. These dialects differ in the number of specific constructs, reserved words, comment style, or whether nested comments are allowed. We solve these differences in the parser: Our grammar supports a superset of all dialects. During parsing one can switch—either using a configuration option or using a pragma—between different dialects. This enables or disables certain keywords in the parser so as to allow their usage as identifiers. To exemplify, when parsing Siemens SCL `BEGIN` is a required keyword marking the beginning of the statements after the variable declarations. When parsing source code for other vendors, `BEGIN` is not reserved and should be treated as an identifier. The behavior can thus be selected in our parser (or, more specifically, in our lexer, which filters the keywords/non-keywords appropriately).

Another example is the definition of pointers. The IEC norm allows for defining pointers using the `REF_TO` keyword, whereas CoDeSys uses the keywords `POINTER TO`. We enable and disable these keywords accordingly and then handle pointers under a unified framework.

Finally, our parser is very liberal when it comes to accepting certain syntactic constructs. Semicolons are defined optional at some places (e. g., after control structures) and we allow to omit the closing keywords for function block definitions.

Encrypted / Restricted Code

Another problem we faced analyzing real-world code was that the access to the actual code base was often very restricted. During a case study it turned out, e. g., that some libraries were encrypted and thus not amenable to our analysis [115]. To handle code with unknown or encrypted FBs, we derive the types of unknown variables and function blocks at translation time depending on the context in which they are used. We then try to guess the type of these unknown variables or parameters from the context they are used in. Since input and output parameters are accessed using a different syntax, we are able to distinguish between them. We can hence deduce which variables are affected by calling unknown function blocks and thus still provide a sound over-approximation of the program semantics [115].

For a case study, we collaborated with ABB and implemented a parser for Compact Control Builder AC800M files. The results of this case study are reported in Sect. 8.6.1.

3.4.4 Instructions

Intermediate Representation

Our intermediate representation (IR) is based on a set of primitive instructions which we describe in this section. These instructions allow for expressing all Structured Text, Instruction List and Function Block Diagram programs. They operate on different operands defined as follows:

literal	::=	...	(Literal value)
reference	::=	...	(See below)
lvalue-expr	::=	*variable*	
	|	*negated-variable*	(Only for negated output parameters)
	|	**reference*	(Dereferenced variable)
expr	::=	valueof *lvalue-expr*	
	|	*literal*	
	|	addressof *lvalue-expr*	
	|	$\ominus$ *expr*	
	|	*expr* $\odot$ *expr*	
	|	*expr* $\bowtie$ *expr*	
	|	typecast *expr*	

Here, $\ominus$ denotes a unary operator, $\odot$ a binary or arithmetic operator and $\bowtie$ a relational operator. Intuitively, `lvalue-expr` are expressions which a value can be assigned to, whereas all `expr` have a value. References are special variables created internally to reference to array elements or structure fields. They can also be used to represent pointers for language dialects that support pointers. Typecast expressions are used to convert between different integer and floating point types.

References

We make use of the following intermediate instructions:

- `Assign lvalue-expr, expr`
  This instruction copies the value of `expr` and assigns it to `lvalue-expr`.

- `Alias` `reference`, `lvalue-expr`
  This instruction creates an alias of the expression `lvalue-expr` such that `reference` can be used to refer to the value of `lvalue-expr`. This instruction is used to refer to variables in aggregate data types using the accompanying `Index` and `Member` instructions.

- `TransferFunction` `func-id`, `lvalue-expr`, (`expression`...)
  This instruction calls the internal function `func-id` using the given expressions as operands and assigns the result to `lvalue-expr`. It is used to implement functions not (readily) expressible using the given operations, especially mathematical functions. In its abstract semantics, sound approximations have to be provided for the result (cp. Sect. 3.3).

- `WideningHint` `variable`, `expr`
  The `WideningHint` is a special hinting instruction to improve the runtime of certain analyses. It indicates that `variable` will likely assume values according to `expr`. Using this hint, loop bounds can be inferred faster (cp. Sect. 8.2.7). Since this instruction represents only a hint for speeding up the analysis, it does not have underlying operational semantics.

- `Index` `reference0`, `reference1`, `expr`
  `reference1` must reference an array and `expr` must be a valid index expression into this array. Then, the `Index` instruction will create an alias of the member with number `expr` of `reference1` such that `reference0` refers to it.

- `Member` `reference0`, `reference1`, `field`
  `reference1` must reference a struct and `field` a valid field name of this structured type. Then, the `Member` instruction will create an alias such that `reference0` refers to it.

- `Jump` `label`
  Jump unconditional to `label`.

- `Return`
  End of POU. Always the last instruction in a translation unit, see below.

- `Branch[If/Unless]` `label`, `condition`
  This instruction branches to `label` if/unless the `condition` is fulfilled.

- `Call` `pou`, (operands, ...)
  Call the `pou` instance, passing all input operands provided. After the call, all provided output operands are assigned from the `pou`. Note that functions cannot be called in an expression. A separate `Call` instruction has to be issued for each function call (possibly storing the result in a temporary variable).

- `CallIndirect reference`, (operands, ...)
  This instruction works as the `Call` instruction but the callee is selected via a reference.

Translation Unit

The body of each POU will be reflected by an array of these instructions called *translation unit*. The conditional and unconditional branch instruction can only jump to instructions in this translation unit (the index of the array of instructions is used as the jump target). Other POUs can be invoked using the `Call` or `CallIndirect` instruction only. `Return` is always the last instruction of a translation unit. In case of multiple exits from a POU, jump instruction to the sole `Return` instruction will be generated. For each instruction, we mark the original program statement or fragment it was generated from so as to highlight possible problems in the actual source code. We discuss a complete example of an IL program in Chap. 4, and an ST example in Chap. 8.

Instead of creating one translation unit per POU, we create separate translation units for each instance of a POU: If, e. g., `Block` is an FB with integer variables `x` and `y` and `FB1`, `FB2` are instances of `Block`, then the memory layout looks as follows:

<table>
<tr><td colspan="2">Other variables...</td></tr>
<tr><td rowspan="2"><code>FB1 : Block</code></td><td><code>x : int</code></td></tr>
<tr><td><code>y : int</code></td></tr>
<tr><td rowspan="2"><code>FB2 : Block</code></td><td><code>x : int</code></td></tr>
<tr><td><code>y : int</code></td></tr>
<tr><td colspan="2">Other variables...</td></tr>
</table>

We then create two translation units for `Block`. One that operates on `FB1.x` and `FB1.y`, and one that operates on `FB2.x` and `FB2.y`. Depending on whether the `FB1` or `FB2` instance of `Block` is called, we emit a `Call` instruction to the corresponding translation unit. Alternatively, one could generate one translation unit per POU. This translation unit would then be given a pointer to the POU members on each call on which the code should operate. The advantage of our approach is that each translation unit operates on its own set of variables. While it generates more IR instructions, it greatly simplifies further analyses, since it is statically known to which member variables each instruction refers to.

Decomposition of complex expressions

We use the `Alias` and `Index` instructions to decompose arbitrarily complex expressions with array and structure accesses. To motivate this, we consider the following ST fragment with a double indirect array access:

```
arr0[arr1[a+b]] := arr2[c];
```

This fragment can be modularly decomposed into:

```
Alias ref0, arr1
Index ref1, ref0, a+b
Alias ref2, arr0
```

```
4 Index ref3, ref2, *ref1
5 Alias ref4, arr2
6 Index ref5, ref4, c
7 Assign *ref3, *ref5
```

Observe that in line 4 the value that the variable `ref1` points to is used as an index. In line 7, the actual assignment occurs. Our IR makes it now easy to, e. g., automatically detect possible invalid array access by checking all `Index` instructions, which now have a canonical and simpler form. The true power of this approach becomes evident when handling abstractions of data, i. e., multiple references and values at once during the abstract interpretation in Chap. 8.

4

COUNTEREXAMPLE-GUIDED ABSTRACTION REFINEMENT

In Sect. 2.5.1 of Chap. 2, we described our explicit-state model for PLC programs. Since PLC programs typically depend on several inputs, verification using this explicit-state model is susceptible to state explosion, due to the exponential growth in the number of input bits. Even small programs can easily lead to state spaces consisting of hundreds of millions of states, which is a major obstacle for the applicability of model checking to real-world programs [39]. In this chapter, we describe the process of automatically building an abstracted state space for PLC programs. The approach is based on the counterexample guided abstraction refinement (CEGAR) scheme [40], geared towards the specifics of PLC programs. In a more general setting, these CEGAR techniques have been successfully integrated into several model checkers before [5, 69].

CEGAR

4.1 APPROACH

The key idea of the CEGAR scheme is to start the verification process on a coarse over-approximation of the program semantics. For such an over-approximation, we are trying to prove an ∀CTL formula, the universal fragment of CTL [53] (cp. 2.4.2). These all-quantified specifications have the key property that if they are satisfied in this over-approximating semantics, they are also valid in the concrete model, since the abstraction allows—at most—for more behavior than the concrete system. In case a specification is violated, this may be due to the abstraction, which then manifests itself in a spurious counterexample. In this case, we can use the counterexample to refine the abstraction in order to obtain a stronger semantics which suppresses the behavior that led to the spurious counterexample trace. We tailor our abstraction refinement for the execution model of PLCs as compared to traditional CEGAR approaches implemented in tools such as Slam [5] or Blast [69]: On the one hand, we decide *when* refinement is necessary based on the cyclic behavior of PLCs. On the other hand, *what* is refined (e. g., an input vs. a local variable) can necessitate different strategies for the organization of the state space.

∀CTL

We first detail how refinement is triggered. We, therefore, make use of three indicators: Firstly, we use refinement to ensure deterministic control flow of the

Deterministic Control Flow

program. Traditional CEGAR techniques allow for non-deterministic control flow, i. e., it is possible to have a conditional branch with unknown (abstracted) branch condition. This is not possible for PLCs due to the atomic simulation of a cycle during state space generation. We hence trigger refinement if a conditional branch instruction would yield non-deterministic control flow. The refinement itself will be selected based on the results of a constraint solver on the conditional expression, which, subsequently, will cause a refinement of a program variable. Secondary, our method refines the abstraction on-the-fly if atomic propositions cannot be assigned a truth-value during the simulation of a cycle. This ensures that we can always evaluate the validity of the specification at the end of the cycle. Finally, we use refinement to ensure deterministic behavior of special function blocks (e. g., timers) and instructions that operate on arrays or pointers.

Atomic Propositions

Based on the scopes of variables, our approach utilizes two different methods for *what* is refined: In the case that input variables require refinement, only the currently processed cycle needs to be reanalyzed using the refined semantics. States that evolved from other input combinations are not affected by this refinement step. For variables that endure cycles, however, we use a different approach. As we have seen in Sect. 2.5.1, these are the variables that are non-temporary and not inputs. For exposition, we subsume these variables under the term *local variables* throughout this chapter and denote them by the set $\textsc{Var}_M$ (cp. Def. 2.2). If these local variables trigger the refinement process, the state space admits spurious counterexamples. In this case, a rebuild might be necessary based on globally refined constraints, so-called *lemmas*.

Refinement of Inputs

Refinement of Locals

4.1.1 *Related Work*

Our approach builds on an abstract interpretation framework of program semantics [44, 45], using the domains defined in Sect. 3.3. The techniques in this chapter particular build on the interval domain [44] and bit-wise domain which are combined using a partly reduced product similar to [35, 97, 98]. This allows us to reason about arithmetic as well as bit-manipulating program fragments. In general, our method is inspired by the abstract simulation using intervals described by Schlich et al. [105]. To avoid spurious counterexamples, however, we introduce refinements to our abstraction.

These refinements are based on two principles: Our refinement loop, which starts at a very coarse abstraction that is iteratively refined, is similar to the works of Kurshan [77]. On the one hand, the refinement in this loop is triggered by certain PLC specific behavior, such as deterministic control flow during the execution of a program cycle. On the other hand, we implement the traditional CEGAR-loop [40] that analyzes counterexamples generated by the model checker and—in the case a counterexample is spurious due to the abstraction—can also trigger refinement.

We further use different refinement steps depending on whether we have to refine abstract values on input variables or in local variables. Henzinger et al. [69]

propose a *lazy abstraction scheme* that refines only parts of the predicates in the program. Our refinement step for input variables can be seen as a simplified adaptation of such a method. The tree-based structure for the organization of our state space described in Sect. 4.5 follows similar ideas as described by McMillan [83].

A key difference between our method and existing techniques is that our method exploits knowledge about the underlying PLC semantics to trigger refinement and is not solely based on the analysis of spurious counterexamples.

4.1.2 *Contributions & Outline*

In this chapter, we present the following contributions: We describe a symbolic encoding of programs written in our intermediate representation that we use to derive constraints. The constraints are subsequently used to guide the refinement process. Constraint solving over our interval and bit-wise domain is used during the refinement itself. We detail a CEGAR algorithm that is optimized for refinements based on input and local variables. We show the effectiveness of our method by verifying various function blocks from industry and academia. Using CEGAR, each of these blocks could be verified on a standard desktop computer, requiring less than 2 minutes per block. We will start by presenting our technique on the basis of a worked example, which is written in Instruction List.

4.2 WORKED EXAMPLE

Our approach will be motivated with the example program shown in Fig. 8, which is used throughout the chapter. The program comprises two input variables, a local variable, and an output variable, all of type `BYTE` (range 0–255). In each cycle, the following operation is performed:

- The input variable `input0` is loaded into the accumulator, the constant 50 is added and the result is compared to 100 (lines 11–13).
- If the result is not greater than 100, the input variable `input1` is copied into the local variable `var0` (lines 15–16).
- Otherwise, the local variable `var0` is copied into the output variable `output0` (lines 18–19).

Naïve Method

To verify this program using naïve methods, we would generate the state space as shown in Sect. 2.5.1: We would start by enumerating all possible input configurations, creating all successor states of the initial state. The process is repeated to obtain a state space which can be examined by a model checker. In our example, such an approach would create $2^{16} = 65\,536$ successors for each state, resulting in $2^{32} = 4\,294\,967\,296$ states in total. This is known as the state explosion problem [41], which makes this approach infeasible for larger programs. To make formal verification possible, abstract states have to be introduced, which summarize a (potentially

```
1  PROGRAM Instruction_List_Example
2   VAR_INPUT
3      input0, input1: BYTE;
4   END_VAR
5   VAR
6      var0:           BYTE;
7   END_VAR
8   VAR_OUTPUT
9      output0:        BYTE;
10  END_VAR
11       LD     input0
12       ADD    50
13       GT     100
14       JMPC   lbl
15       LD     input1
16       ST     var0
17       RET
18 lbl:  LD     var0
19       ST     output0
20       RET
21 END_PROGRAM
```

Figure 8: Example Instruction List program [16]

huge) number of concrete states (cp. Sect. 2.5.2). In the given program, e. g., it is only relevant whether `input0` lies in the interval $[0, 50]$ or in $[51, 255]$ to determine the two possible control flow paths: If we determine that the value lies in either interval, we can decide if the conditional jump in line 14 is taken or not (independent of the accumulator or other variables).

The crucial step in this method is to find abstract values that do not change the behavior of the program or only change it when it is irrelevant for the evaluation of the specification. To find such abstract values, we gradually refine the abstraction: We start with the most general abstract states representing all possible values. Then, these values are successively refined as long as the program behavior is different to the original behavior w. r. t. the specification.

In the example program, we would assume the abstract value $[0, 255]$ for inputs `input0` and `input1` (i. e., all values are possible) and start simulating the PLC cycle: After loading `input0` and adding 50, the accumulator holds $[50, 305]$ (no overflow occurs here, since the accumulator can store larger data types than bytes). Comparing the latter interval to 100 yields the set $\{true, false\}$ in the accumulator, because the comparison can result in either *true* or *false* depending on the actual values assumed in a concrete execution. The next operation is a conditional jump, for which we hence cannot decide whether it would be taken. Since a PLC cycle is executed atomically, we do not want to split the execution into two different paths here and thus use this as a refinement criterium: We always demand that the accumulator holds a concrete value before a branch. Each conditional jump thus poses a restriction on the abstract value in the accumulator.

From Restrictions to Constraints

We call such a restriction a *constraint*. The key idea is to use the constraint on the abstract value in the accumulator in line 14 to derive a constraint on the input variables that caused the conditional jump to be ambiguous. The reason the accumulator contains $\{true, false\}$ is the comparison of $[50, 305]$ with 100. It is, therefore, sufficient to constrain the interval $[50, 305]$ to be either greater than 100 or less-equal than 100. Since the interval $[50, 305]$ was the result of adding 50 to $[0, 255]$, we can constrain $[0, 255]$ to be either greater than 50 or less-equal than 50 to avoid conflicts. We observe further that the interval $[0, 255]$ is the initialization of the variable `input0`, so we now derive that `input0` has to be considered for two different initializations: the intervals $[0, 50]$ and $[51, 255]$, which both ensure deterministic control flow and cover the whole range of `input0`. This process of resolving constraints from intermediate expressions to input variables will later be performed using symbolic information.

By careful observation, we resolved the constraint on the accumulator for the conditional jump to a constraint on an input variable in this case. Then, by refining the input variable into two different intervals, problematic combinations of values are avoided in subsequent executions after restarting the cycle. A key observation at this stage is that the values of input variables are assigned independently of previous states, and thus, the refinement does not affect already created states. Hence, constraints on input variables can be resolved locally by splitting the abstract values into smaller abstract states. If, however, we have a situation where we have constraints on local variables such as `var0`, we have to use a different strategy. Here, splitting the abstract value cannot be resolved locally, because its value was calculated in a previous state and we no longer know how this value was derived symbolically. In Sect. 4.4.2, we will detail how we can handle this situation.

First, we will introduce the constraint solver, which is used for the constraint transformation process on symbolic information during the execution of a program cycle. We then formally present the refinement process for input and local variables using the constraint solver.

4.3 CONSTRAINT SOLVER

A constraint is a certain condition, which can either be applied to an abstract value or a symbolic expression. The constraint solver is then used to transform constraints on symbolic expressions into constraints on variables containing abstract values, ideally equivalent to the original constraint. The rationale is that resolving a constraint for an abstract value is trivial (by selecting suitable values) whereas resolving constraints on an expression is more convoluted.

4.3.1 *Constraints on Abstract Values*

Constraint $\mathsf{cs}_f(a)$

Definition 4.1: A *constraint* is a condition f on an abstract value a, denoted $\mathsf{cs}_f(a)$. We call such a constraint *valid* or *consistent* if the set of concrete values that a represents is consistent under the condition defined by f. We define the different constraints, with the meaning of consistent as follows:

- The single value constraint $\mathsf{cs}_{sing}(a)$ is consistent iff a represents only a single concrete value.
- Comparison constraints have the form $\mathsf{cs}_{\bowtie c}(a)$ for some relational operation $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$ and a constant c. They are consistent iff for all $x, y \in a$ the condition $x \bowtie c \iff y \bowtie c$ holds.
- The bit mask constraint $\mathsf{cs}_{\&\, c}(a)$ is consistent iff for all $x, y \in a$: $x \,\&\, c = y \,\&\, c$, where & is the bitwise AND operation.

Concrete Values

A constraint can be seen as a restriction on *how many* concrete values an abstract value can represent at most without getting inconsistent. Trivially, concrete values are consistent under all constraints. It follows that all constraints can be made consistent by splitting an abstract value into the concrete values it represents.

Selecting Consistent Values

Given a constraint and a variable, we can easily assign abstract values to the variable such that the constraint is valid. Usually, we want these abstract values to cover as many concrete values as possible. This is done using a splitter:

Splitter

Definition 4.2: Let $\mathcal{A}$ be an abstract domain, $a \in \mathcal{A}$ an abstract value and f a constraint condition. A *splitter* $\mathsf{split}_f : \mathcal{A} \to \wp(\mathcal{A})$ enumerates abstract values $\mathsf{split}_f(a) = \{a_1, \ldots, a_n\}$ such that $\dot{\bigcup}_{i=1}^{n} a_i = D$ and $\mathsf{cs}_f(a_i)$ is consistent for $1 \leq i \leq n$ with n minimal.

To illustrate, let $\mathsf{cs}_{>100}([0, 255])$ be a constraint on a BYTE interval. In this case, the splitter $\mathsf{split}_f([0, 255])$ would generate the consistent abstract values $[0, 100]$ and $[101, 255]$. That means that for a variable v of type BYTE, assigning both of these values to v makes $\mathsf{cs}_{>100}(v)$ valid. Note that we extended the definition of constraints from abstract value to variables here.

Constraints on Variables

We can hence conclude that once we have derived a constraint on a variable, the variable can be easily (and efficiently) made consistent by a splitter. Constraints on arbitrary expressions have to be reduced to constraints on variables. In our example, this step was the transformation of the single value constraint (which arises from the conditional jump) to the compare constraint on the variable `input0`. To formalize this process, we extend the definition to constraints of expressions of abstract values, written $\mathsf{cs}_f(expr)$.

Program	Intermediate	Symbolic form	Accumulator
`LD input0`	`Assign ACC, input0`	$\mathsf{acc}^{(0)} := \mathsf{input}_0{}^{(0)}$	$[0, 255]$
`ADD 50`	`Assign ACC, (ACC + 50)`	$\mathsf{acc}^{(1)} := \mathsf{acc}^{(0)} + 50$	$[50, 305]$
`GT 100`	`Assign ACC, (ACC == 0)`	$\mathsf{acc}^{(2)} := \mathsf{acc}^{(1)} > 100$	$\{true, false\}$
`JMPC label`	`BranchUnless ACC, label`	$\mathsf{guard}(\mathsf{cs}_{sing}(\mathsf{acc}^{(2)}))$	
`..`			

Figure 9: IL fragment, first translated to IR then into SSA form. The rightmost column shows the values in the current accumulator in one execution path.

4.3.2 *Constraints on Expressions*

Constraints on expressions are derived from the intermediate representation (IR) we introduced in Sect. 3.4.4. To derive expressions, we rewrite instructions of our IR into a *static single assignment* (SSA) form [47] on which the constraint solver will operate. Since this translation performed while simulating one cycle, only one path of the program is considered at a time. Therefore, we do not have to generate φ-nodes in the translation phase[1]. In case an expression results in a concrete value during the build of the SSA, we discard the symbolic information and use the concrete value as a right-hand side. This prunes unnecessary information and ensures that all non-constant expressions are composed of at least one variable that can be refined.

Static Single Assignment

Figure 9 shows an example of how the translation is performed. On the left hand side, the example program is shown and then its translation into IR (cp. Fig. 8). The third column shows the corresponding SSA expressions. In the forth column we list the abstract values of the accumulator for the first instructions of the example program. The `LD`, `ADD`, and `GT` instructions of IL are translated into `Assign` instructions in our IR. Translating such assignments into SSA is straightforward by introducing a new variable for each left hand side. On the right hand side, however, complex expressions are allowed in our IR (which arise, e. g., when translating ST programs). For the purpose of the constraint solver, each complex expression is converted into simpler expressions by introducing temporaries[2]. This ensures that unary and binary expressions (cp. Sect. 3.4.4) only operate on l-values or constants.

For the `BranchUnless` statement, we need to decide the conditional jump and thus require a concrete value in the accumulator. Therefore, `guard` statements are added, which contain the appropriate constraints. If these constraints are inconsistent, the constraint solver described in the next section is used to find refinements of variables in order to make the guard constraint valid.

1 φ-nodes are used in SSA to merge two different incoming control flow edges. This cannot happen in our approach because we only consider one path at a time.

2 Similar to a three-address code.

Note that the translation into SSA is done during the simulation, automatically unrolling all loops of the program. Since the programs of PLCs should react fast to ensure their real-time behavior, each program cycle should terminate after a short time, which guarantees bounded size of these symbolic expressions. In the next section, we examine how expression constraints such as $\mathsf{cs}_{sing}(\mathsf{acc}^{(2)})$ are transformed.

4.3.3 *Transforming Constraints*

Implication of Constraints

If the validity of an expression constraint $\mathsf{cs}_{f_2}(e_2)$ implies the validity of $\mathsf{cs}_{f_1}(e_1)$, we write $\mathsf{cs}_{f_1}(e_1) \curvearrowright \mathsf{cs}_{f_2}(e_2)$. We illustrate this using the example program: Consider the single-value constraint $\mathsf{cs}_{sing}(\mathsf{acc}^{(2)})$. From this constraint, the solver can derive a constraint on `input0` with the following steps:

$$\begin{aligned}
\mathsf{cs}_{sing}(\mathsf{acc}^{(2)}) &\curvearrowright \mathsf{cs}_{sing}(\mathsf{acc}^{(1)} > 100) && (1)\\
&\curvearrowright \mathsf{cs}_{>100}(\mathsf{acc}^{(1)}) && (2)\\
&\curvearrowright \mathsf{cs}_{>100}(\mathsf{acc}^{(0)} + 50) && (3)\\
&\curvearrowright \mathsf{cs}_{>100-50}(\mathsf{acc}^{(0)}) && (4)\\
&\curvearrowright \mathsf{cs}_{>50}(\mathsf{input}_0{}^{(0)}) && (5)
\end{aligned}$$

In the trivial steps (1), (3) and (5) left-hand side of an SSA expression is replaced by its corresponding right-hand side definition. Step (2) transforms the single-value constraint into an equivalent compare constraint. In step (4), a compare constraint is translated to resolve adding the constant. To summarize, we can make the single-value constraint $\mathsf{cs}_{sing}(\mathsf{acc}^{(2)})$ valid by refining `input0` into proper abstract values using a splitter on $\mathsf{cs}_{>50}(\mathsf{input}_0 0)$.

We formally define the steps of the constraint solver inductively on the SSA expressions. In the following, f is an arbitrary constraint condition, e_1 and e_2 are (non-constant) expressions, c is a constant, $\ominus$ is a unary expression and $\odot$ is a binary operation. Relational operations are denoted by the symbol $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$.

Let $\mathsf{cs}_f(e_0)$ be a constraint. If e_0 is an l-value and there is an SSA expression $e_0 := e_1$, we apply the trivial transformation $\mathsf{cs}_f(e_0) \curvearrowright \mathsf{cs}_f(e_1)$. If e_0 is an input variable or a local variable the resolving process is terminated, since this can be handled using a splitter. If e_0 is constant, it is trivially consistent under all constraints.

Unary Expressions

Otherwise, e_0 is a unary expression, a binary expression or a data type cast. For a unary expression, the transformation is defined as follows:

- A complement operation is absorbed by a bit-mask constraint $\mathsf{cs}_{\&\, m}(\neg e_1) \curvearrowright \mathsf{cs}_{\&\, m}(e_1)$.
- A compare constraint on a negation $\mathsf{cs}_{\bowtie c}(-e_1)$ is resolved by $\mathsf{cs}_{\bowtie c}(-e_1) \curvearrowright \mathsf{cs}_{\overline{\bowtie} -c}(e_1)$, where $(\overline{=}, \overline{\neq}, \overline{<}, \overline{\leq}, \overline{>}, \overline{\geq}) = (=, \neq, \geq, >, \leq, <)$.

- All other constraints on unary operations are resolved as single value constraints $\mathsf{cs}_f(\ominus e_1) \curvearrowright \mathsf{cs}_{sing}(e_1)$.

For a binary expression the transformation is defined as follows: *Binary Expressions*

- A constraint on two non-constant expressions is resolved as a single value constraint on one expression $\mathsf{cs}_f(e_1 \odot e_2) \curvearrowright \mathsf{cs}_{sing}(e_1)$. This other expression is then resolved in the next refinement step.
- For all compare operations $\bowtie$ we resolve $\mathsf{cs}_{sing}(e_1 \bowtie c) \curvearrowright \mathsf{cs}_{\bowtie c}(e_1)$.
- Addition and subtraction in compare constraints are resolved by the translations $\mathsf{cs}_{\bowtie c_1}(e_1 + c_2) \curvearrowright \mathsf{cs}_{\bowtie(c_1 - c_2)}(e_1)$, $\mathsf{cs}_{\bowtie c_1}(e_1 - c_2) \curvearrowright \mathsf{cs}_{\bowtie(c_1 + c_2)}(e_1)$, and $\mathsf{cs}_{\bowtie c_1}(c_2 - e_1) \curvearrowright \mathsf{cs}_{\bowtie(c_1 - c_2)}(-e_1)$.
- Some bitwise operation are resolved using the bit mask constraint. A comparison constraint $\mathsf{cs}_{=c_1}(e_1 \& c_2)$ is transformed as $\mathsf{cs}_{=c_1}(e_1 \& c_2) \curvearrowright \mathsf{cs}_{\& c_2}(e_1)$.
- All other constraints on binary operations are resolved as single-value constraints: $\mathsf{cs}_f(e_1 \odot c) \curvearrowright \mathsf{cs}_{sing}(e_1)$. A constraint $\mathsf{cs}_{\neq c_1}(e_1 \& c_2)$ is transformed similarly.
- Casts to a smaller data type are handled similar to bit mask constraints, while casts to larger data types are ignored (since they do not change semantics).

Since each IL instruction adds at most one SSA expression, the constraint solver can resolve each constraint in, at most, $\mathcal{O}(n)$ steps, where n is the number of instructions executed in the cycle. For ST programs, n is the number of operations, since each operation adds another SSA expression. In the next section, we will use the constraint solver with linear complexity to compute the necessary refinements. *Complexity*

4.4 REFINEMENTS

Existing CEGAR techniques work solely by analyzing counterexamples. Spurious counterexample then trigger refinements of the chosen abstraction. A key difference of our approach to these techniques is that we take PLC specific behavior into account, which yields additional hints for refinements. We want to prevent, e. g., non-deterministic control flow during the simulation of a program cycle so as to hide intermediate states, since they should not be observable to the model checker.

At each non-deterministic branch point, we therefore symbolically resolve the non-deterministic value to the source that generated the value. The source is usually an input variable (whose value is chosen non-deterministically each cycle) but could also be a local variable or a timer. Splitting the abstract value of such variables into different (smaller) abstract values creates separate states, and can thus eliminate the problematic cases.

Constraints during Simulation

A refinement of values using the constraint solver is initiated once an inconsistent constraint is encountered. Therefore, the validity of constraints is tested during simulation. We introduce constraints (by the means of guard instructions) in the following situations:

- As we have seen, the control flow has to be deterministic while simulating a cycle. Hence, we put a `guard` before each `BranchUnless` instruction, which contains the branch condition as a single value constraint.
- Some special function blocks such as timers require concrete input values for their operation, which we guard accordingly.
- After simulating a cycle, the truth valuations of atomic propositions are determined to label the state space accordingly. The values of the atomic propositions have to be consistent, so they are guarded with appropriate constraints.
- The `Index` instruction is guarded by single value constraints, because we do not allow indirect array access with an unknown index.
- Similarly, all pointer instructions are guarded by single value constraints.

Two Strategies

Our method follows two different strategies, depending on whether an input or a local variable has to be refined. In the first step we will explain how we implemented the refinement of input variables.

4.4.1 *Refinement of Input Variables*

We will first consider the case that the refinement algorithm does not have to refine values stored in predecessor states. This is achieved by allowing only concrete values in local variables (i. e., variables whose value is retained between cycles) at the start and at the end of each cycle. We can easily achieve this using the existing approach by guarding these variables with a single-value constraint at the end of the program. This ensures that only input and temporary variables can store an abstract value at the beginning and end of the cycle. Since the value is overwritten in the next cycle, we guarantee that no abstract values are maintained between states.

Since this technique forbids abstract values in the state space, we do not add additional behavior to the program, and hence, will not find spurious counterexamples. We will see that the refinement of input variables is a powerful abstraction of the state space on itself, due to the huge number of hidden input values. Our algorithm iteratively refines values, similar to the refinement loop initially described by Kurshan [77]. It performs the following steps during the generation of successor states (cp. Def. 2.4 and the following discussion):

1. All splitters used for the refinements are stored on a stack. In the first step, a splitter is pushed onto the stack that assigns the $\top$ element of the domain to all input variables.

2. The splitter on top of the stack is used to assign abstract values to the input variables. The splitter gives rise to different configurations $s_1, \ldots, s_n$ (representing different traces through the program). For $s_1 \ldots s_n$, steps 3–6 are performed:

3. Simulate a cycle of the PLC for the current configuration. If one of the above mentioned situations occurs, where the simulation cannot proceed, the constraint solver is used to find a new splitter, which is then put on the stack. In this case, step 2 is repeated.

4. The atomic propositions are evaluated. If a truth value cannot be determined, again, the constraint solver is used to find a new splitter, which is put on the stack and step 2 is repeated.

5. The newly created successor state is stored in the state space.

6. The splitter on top of the stack is advanced to its next refinement. If the splitter has already assigned all values of the domain, it is removed from the stack. If the stack is empty all successors are created. Otherwise repeat with step 2.

Note that by using a stack for the splitters, we work with different splitters depending on the current assignment to variables. This also implies that we can use different splitters, depending on the program path that is currently refined (since this path is determined by the current assignment to the program variables).

It follows that the efficiency of this approach is highly dependent on the order in which variables are refined. Typically, variables are referenced in the order of their importance for the control flow in real-world programs. Hence, the refinements picked by of our approach are usually quite good. In the next section, this method is extended to other variable classes.

4.4.2 *Refinement of Local Variables*

We now discuss how the algorithms works if we allow for storing of abstract values in local variables. Since the value of local variables might be calculated in a previous state (and thus depend on the value of other variables), abstract values in local variables can incur new behavior, i. e., transitions which are not possible in the concrete model. To illustrate, consider two variables that contain abstract values, but with their concrete value always being identical in the concrete program semantics. In the example program, this is the case for the variables `var0` and `outputp0` if `input0` is greater than 50.

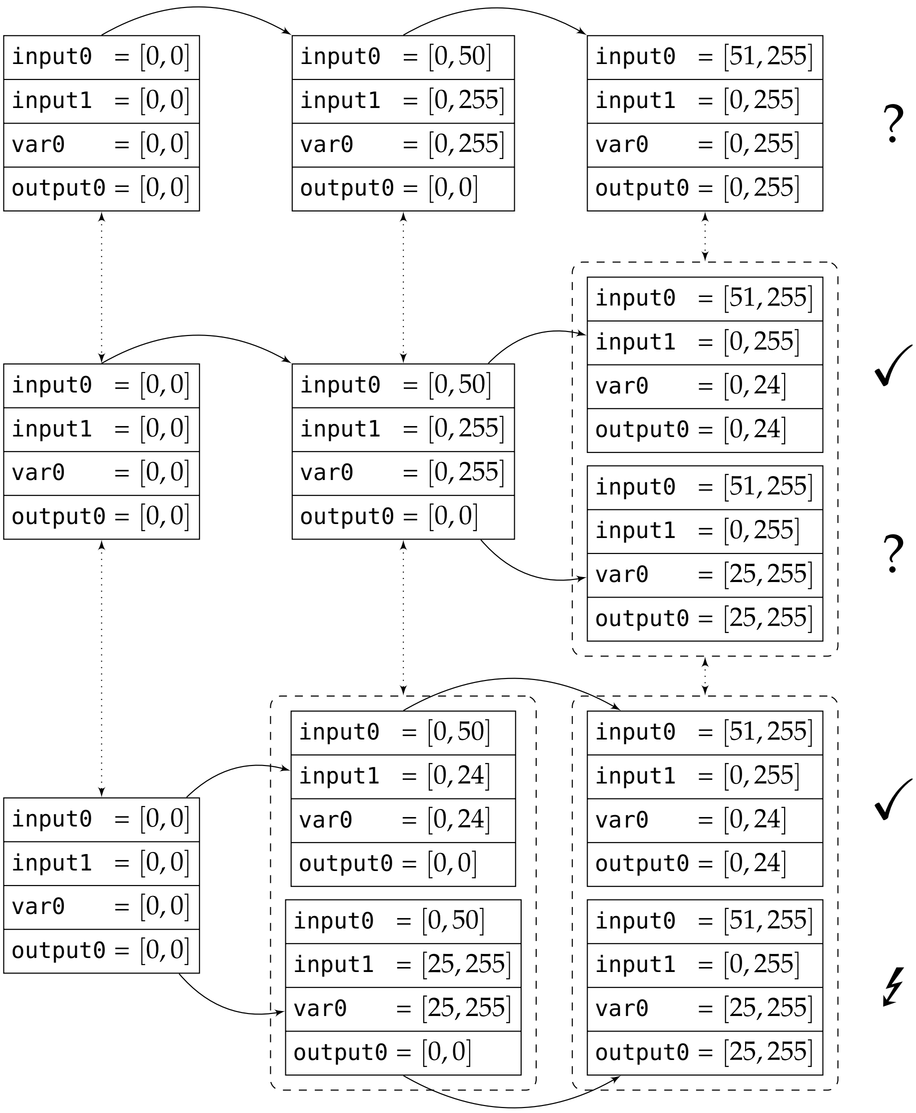


Figure 10: Subsequent refinements of the state space for verifying the specification AG `output0` < 25 [16]

Since we work with non-relational domains, we cannot track that two variables are identical if they contain an abstract value. When comparing the value of the variables, we hence induce new behavior since we assume that they could contain different values. Yet—and this is the key property—if an $\forall$CTL formula is valid in such an abstract model with added behavior, it is also valid in the concrete model [40]. Otherwise, the formula is violated and the model checker is able to extract a counterexample. A counterexample that is not feasible in the concrete semantics is called *spurious*. Our approach to verify that a counterexample is not spurious is to rebuild the state space based on a refined semantics.

The key steps of this approach are sketched in Fig. 10. We are trying to verify AG `output0` < 25 for the example program. The first row shows the first iteration of the state space, with irrelevant states omitted. In the right-most state, $\mathtt{cs}_{sing}(\mathtt{output0} < 50)$ is not consistent since `output0` lies in the interval $[0, 255]$, so the state has to be refined accordingly. We perform this step using the constraint solver, which returns the constraint $\mathtt{cs}_{>25}(\mathtt{var0})$. The reason for this is that if `input0` lies in the interval $[51, 255]$, `var0` is copied into `output0`. Since refining the global variable `var0` possibly creates new behavior, we save the constraint $\mathtt{cs}_{>25}(\mathtt{var0})$ as a so-called lemma for further refinement:

Definition 4.3: A *lemma* is a constraint (on a local variable) that has to be consistent for a counterexample to stay feasible. In other words, by keeping a lemma consistent during state space generation, we could be able to suppress a spurious counterexample. *Lemma*

In the second row we show the refined state space, where the state was split to make the atomic propositions consistent. Since `output0` > 25 in the state next to the question mark, we have a candidate for a counterexample trace here. Due to the over-approximation, however, we have to verify that this counterexample is also feasible in the concrete semantics. To achieve this we rebuild the state space while keeping all lemmas we found consistent, thus avoiding the addition of new behavior to the state space.

We, therefore, add new guards for all local variables at the end of the program according to their lemmas. The idea is that we can use the symbolic information from the SSA and the end of the program for all variables accessed in this cycle. Hence, the constraint solver can deduce how the value was computed and select suitable refinements. It will either obtain a crucial refinement of an input variable, thus resolving the over-approximation in this state, or it will obtain a new lemma, which might be needed in a further refinement/rebuild step.

The final result of the state space is shown in the third row of Fig. 10. Here, $\mathtt{cs}_{>25}(\mathtt{var0})$ is consistent and all additional behavior was removed. Hence, we can deduce that the counterexample trace is a feasible counterexample for the formula AG `output0` < 25.

4.5 STATE SPACE ORGANIZATION

The organization of the state space described in the previous section has two drawbacks: Firstly, in the case of a spurious counterexample the complete state space has to be rebuilt. Secondly, it does not exploit the structure of the relation between abstract and concrete states. To exemplify, the states $s_0 = \langle v \mapsto 0 \rangle$, $s_1 = \langle v \mapsto [0,5] \rangle$ and $s_2 = \langle v \mapsto [0,10] \rangle$ are currently different states and thus stored independently. Yet, $s_0 \sqsubseteq s_1 \sqsubseteq s_2$ and hence it might be sufficient to only store s_2.

We will, therefore, organize the state space in a way that such entailments can be efficiently detected and their structure exploited. For this, we use a hierarchical representation in the form of a tree. Each node in the tree can either be (a) a constraint $\mathsf{cs}_f(v)$, (b) a *leaf*, or (c) *missing*. Leaves correspond to the abstract states of our state space. Only a constraint $\mathsf{cs}_f(v)$ can have children, defined according to their splitter:

Tree and Nodes

- For a single value constraint $\mathsf{cs}_{sing}(v)$ a child node is introduced for every possible value of v.
- For a comparison constraint $\mathsf{cs}_{\bowtie c}(v)$ two child nodes are introduced for $v \bowtie c$ being true or false, respectively.
- A mask constraint $\mathsf{cs}_{\& c}(v)$ has child nodes consistent to $\mathsf{cs}_{\& c}(v)$. The number of children is thus 2^n, where n is the number of bits set in c.

The key idea here is that the children of a constraint are always consistent under all parent constraints.

Lookup

A *lookup* maps a state s to a state $\hat{s}$ (i. e., a leaf) which is contained in the state space and entails s. A lookup is performed by recursively traversing the tree until a leaf is found. We start at the root and for each node n:

- If n is a leaf $\hat{s}$ then we return $\hat{s}$.
- If n is missing, we have to create a new leaf: We create a new state $\hat{s}$ consistent under all constraints on the path from the root to n and return $\hat{s}$.
- If n is a constraint $\mathsf{cs}_f(v)$ and it is inconsistent under s, we return *refine using the constraint* $\mathsf{cs}_f(v)$. In this case, the lookup fails and the input has to be refined first.
- If n is a constraint $\mathsf{cs}_f(v)$ and it is consistent under s, we continue with the respective child node, where f is fulfilled.

Initial State Space

To build the initial state space for a formula φ, we perform the following operation for the initial state s_0 (i. e., the concrete initial state).

1. We perform a lookup of s_0, which gives rise to a leaf $\hat{s}_0$. Observe that s_0 is a concrete state so this lookup cannot be inconsistent.

2. If the obtained $\hat{s}_0$ is consistent under φ then $\hat{s}_0$ is returned as the initial abstract state.
3. Otherwise, we use the constraint solver to generate a constraint $\mathrm{cs}_f(v)$ resolving this conflict. We then replace the leaf $\hat{s}_0$ by this constraint and resume at step 1.

Building the Abstract State Space

After we have built the initial state space, the abstract state space can be generated. Therefore, we create the successors $s_1, \dots s_n$ of the initial state $\hat{s}_0$ using the refinement techniques described in Sect. 4.4.1 and 4.4.2. Then, the states $s_1, \dots s_n$ are looked up in the state space which gives rise to further refinements (if the lookup is inconsistent) or new leaves (if a node is missing). This process yields thus a transition relation between the leaves of the state space. For each new leaf, the steps are repeated until the complete state space is build or a counterexample is found.

4.5.1 *Counterexample Analysis*

Spurious Counterexamples

If a counterexample is found, it might again be spurious. We hence have to analyze counterexamples for their feasibility and—if they are spurious—add new constraints to suppress them. Further, we want to make counterexamples more explicit by adding back the variables they depend on since their abstract counterpart typically does not explain the violation. In the following, let $\pi = \langle \hat{s}_0, \dots, \hat{s}_n \rangle$ be a counterexample for some invariant φ. We then walk the counterexample backwards. For each transition $\hat{s}_i \to \hat{s}_{i+1}$ we check whether the transition depends on a lemma on some local variable v. If it does, the lemma gives rise to a new constraint $\mathrm{cs}_f(v)$: If the constraint condition is not fulfilled, the counterexample π becomes infeasible. To make the constraint always consistent, we hence replace the leaf $\hat{s}_i$ with $\mathrm{cs}_f(v)$ in the state space. All children of $\mathrm{cs}_f(v)$ will now (by definition) be consistent under this constraint. We now have to rebuild parts of the state space. If $i = 0$ then the initial state $\hat{s}_0$ is changed and we have to rebuild the state space from scratch to check for another counterexample. If $i \neq 0$ then we recheck φ beginning at $\hat{s}_{i+1}$ to test whether the suffix of π is suppressed.

This process is continued until we either no longer find a counterexample (in this case the state space is safe) or we find a counterexample without dependence on lemmata. This is then a feasible counterexample and presented to the user. To make the counterexample more understandable, we also augment it with all lemmata on input variables.

4.5.2 *Worked Example*

We explain our approach using the worked example shown in Fig. 11, which sketches a safety function block that has 4 inputs and 2 outputs. The block operates in different modes. If the block is in output mode, the `input` is copied into

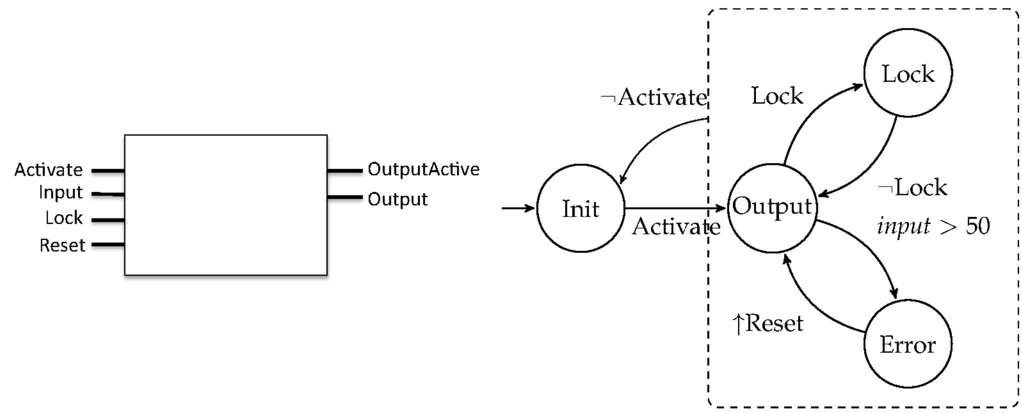
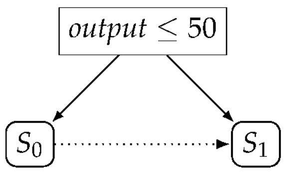


Figure 11: Example function block we use to demonstrate the state space organization.

`output`. If `input` is greater than 50 the block goes into an error state, from where it is necessary to reset the block. Additionally, the output can be locked, in which case the input is no longer copied. Internally, the block is implemented as a state machine with states *Init (0)*, *Ouput (1)*, *Lock (2)*, and *Error (3)*. The safety function we want to verify is:

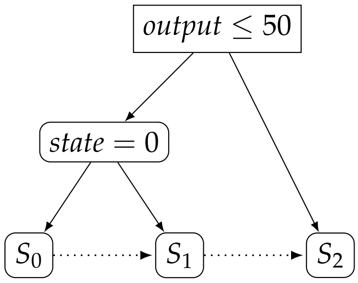
$$AG\ \texttt{output} \leq 50$$

We start by making the initial state space consistent. This entails creating a splitter for $\mathsf{cs}_{sing}(\texttt{output} \leq 50)$. Checking this state space creates a counterexample $S_0 \to S_1$:

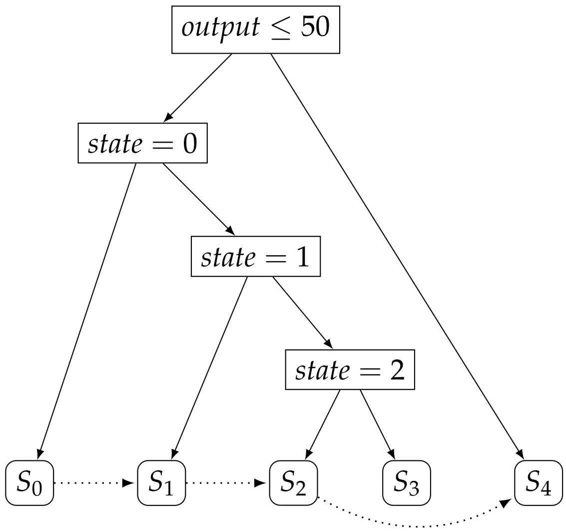


This state space comprises the node $\mathsf{cs}_{sing}(\texttt{output} \leq 50)$, which has the leaves S_0 and S_1. S_0 represents states where $\texttt{output} \leq 50$, which is in particular the initial state. The leaf S_1 represents the violating states. These states are, as we will see later, unreachable. The dotted line represents the counterexample, while the solid lines represent the lookup tree of the state space. The transition of the counterexample depends on the lemma $\texttt{state} = 0$. We hence replace S_0 by this lemma.

Checking the state space again results in the following counterexample:

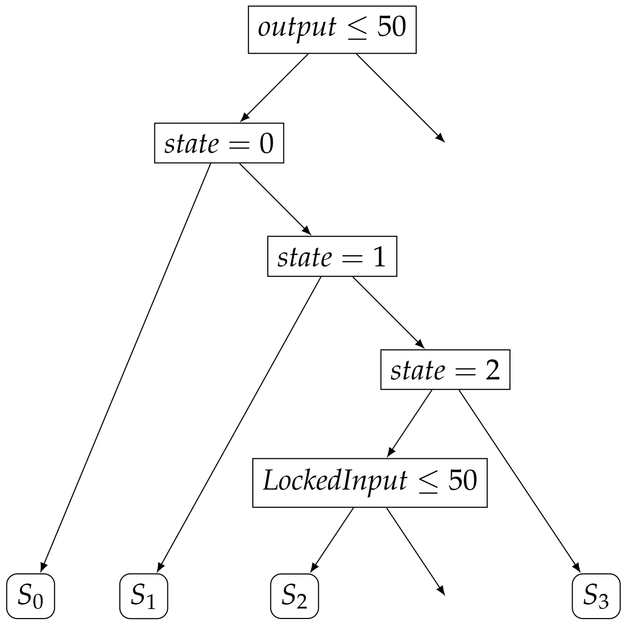


Observe that in S_0 we have `state` $= 0$, while in S_1 we have `state` $\neq 0$. In S_2 we have a violation, which depends on the lemma `state` $= 1$. We use this as a replacement for S_1. Checking the state space again results in a spurious counterexample (not shown), which now depends on `state` $= 2$. Checking the refined state space results in the following spurious counterexample:



This counterexample has a transition from S_2 (where `state` $= 2$, which represents the locked state) to a violating state. This happens because the implementation copies the internally stored value to the output if it is in the locked state. This internal variable is called `LockedInput`. Since our state space does not store any information about this variable, it might contain any value, hence causing the violation. Thus, the final lemma that we derive from this state space is `LockedInput` ≤ 50, which is used to replace S_2. *`LockedInput`*

After a rebuild, we end up with the following state space:



The final state space has only 4 leaves, which represent the four different modes of the function block. Implicitly, it is derived that always `LockInput` ≤ 50 if the block is in state 2 (this follows because the other leaf of the node is missing). Since the right leaf of the `output` ≤ 50 constraint is missing as well, it follows that a violation of the property we want to verity is not reachable.

Violation is Unreachable

4.6 CASE STUDIES

Setup

We have implemented the techniques described in this chapter in ARCADE.PLC. To show the effectiveness of our approach, we performed a case study verifying different properties of various PLC programs [18]. It contains three different sets of programs: We analyze five safety function block from the PLCOPEN consortium [94] from two libraries: The first library is written in IL and was provided by Soliman and Frey [114], while the second library is our own implementation written in ST. The blocks have between 4 and 12 inputs of type `BOOL` and `TIME`.

The second set of programs are written in the Siemens dialect of IL. They are written for a Siemens SIMATIC S7 PLC to control conveyor belts and a 3D robot of a Fischertechnik plant. The conveyor belts operate independently using motors and light curtains. To assess the scalability, we used programs to control one as well as four belts. The robot has three motors to move its arm and one motor for a mechanical grab. The motors are monitored using step counters. We checked the programs with one and four degrees of freedom.

Verification

For the `Antivalent` block we verified that (1) `AntivalentOut` output implies that `ChannelNC` (normally closed) is set and `ChannelNO` (normally open) is not set. To

	Program	φ	res	Abs.	#States	#Created	#CE	Time
PLCOPEN	`Antivalent` (ST, 108 LOC)	(1)	✓	A_1	45	401	0	0.1 s
				A_2	5	782	4	0.1 s
	`EmergencyStop` (IL, 226 LOC)	(2)	✓	—	134	4 288	–	0.4 s
				A_1	80	721	0	0.2 s
				A_2	6	329	7	0.2 s
	`ModeSelector` (IL)	(3)	✓	A_1	> 35 000	> 15 M	0	> 1 h
				A_2	38	623 352	0	55 s
	`ModeSelector` (ST, 187 LOC)	(3)	✓	A_1	15 594	3 254 198	0	270 s
				A_2	17	4 722	0	0.7 s
	`ModeSelector` (ST, 187 LOC)	(3)	✓	A_1	15 594	3 254 198	0	274 s
				A_2	18	4788	2	1.1 s
	`GuardLocking` (IL, 321 LOC)	(4)	✓	—	> 130 000	> 7 M	0	> 1 h
				A_1	39 231	953 741	0	192 s
				A_2	3	1146	0	0.2 s
	`MutingSeq` (ST, 212 LOC)	(5)	✓	A_1	56 466	5 003 709	0	715 s
				A_2	3	1 062	0	0.1 s
	`MutingSeq` (ST, 212 LOC)	(6)	✓	A_1	98 725	16 697 622	0	3 262 s
				A_2	3	1 764	0	0.2 s
BELT	1 Belt (S7 IL, 92 LOC)	(7)	✓	—	360	46 081	–	2 s
				A_1	109	1 448	0	0.2 s
				A_2	3	33	0	0.1 s
	4 Belts (S7 IL, 322 LOC)	(7)	✓	A_1	118	1 579	0	0.4 s
				A_2	3	33	0	0.1 s
ROBOT	1 Axis (S7 IL, 65 LOC)	(7)	✓	—	173	693	–	0.5 s
				A_1	128	371	0	0.5 s
				A_2	208	1026	335	16.6 s
	4 Axes (S7 IL, 101 LOC)	(8)	✓	—	11 921	3 051 777	–	142 s
				A_1	149	473	0	0.5 s
				A_2	166	582	692	201 s
	4 Axes (S7 IL, 101 LOC)	(9)	↯	—	11 665	83 969	–	8.7 s
				A_1	107	237	1	0.5 s
				A_2	82	328	675	261 s

Table 3: Evaluation of the CEGAR technique to verify PLC programs (see also [18])

check the `EmergencyStop` block, we verified that the emergency output is not set if the emergency input is not set (2). For the `ModeSelector` block we verified that in the locked state, at most, one mode is set (3). For the `GuardLocking` block we verified that the `GuardLocked` signal can only be asserted if the block is ready (4). Similarly, we verified that `MutingActive` signal can only be asserted if the `MutingSeq` block is ready (5), and the muting lamp is on (6). For the conveyor belt program of the Fischertechnik plant we verified that it acknowledges the motor stop signal (7). For the robot program we verified that the counter for axis 0 stays in its bounds (8). Otherwise, the plant could suffer physical damage. Additionally, we slightly modified the formula to induce a counterexample (9).

Results

The results of this case study are presented in Tab. 3. We tried to verify each property without abstraction (—), with refinements but without the state space organization (A_1), and finally with full abstraction (A_2). For each technique, the table shows the number of abstract states in the final state space, the number of states created, the number of analyzed counterexamples and the overall time for model checking. Often, we were unable to verify the property without abstraction, since the program depends on too many inputs. For brevity, we only show the results with activated abstractions in these cases.

Using the right abstractions, all formulae for all programs could be verified. The runtime of the verification process is always between seconds and minutes depending on the abstraction selected. Without abstraction, we were unable to verify most programs. The `Antivalent` block, e. g., is one of the smallest programs we checked, but due to one input of type `TIME`, we could not enumerate all input configurations in a concrete domain. The use of abstractions makes this block then amenable to the verification.

The Belt example shows that adding independent functionality (i. e., adding three independent belts to the program) does not affect the abstractions: The same number of abstract states for A_2 is generated. While this works in the Robot example as well, it becomes apparent that the higher number of inputs makes the exploration of this state space slower, even if it is very small. Interestingly, the A_2 abstraction is generally slower in this example. This is caused by the high number of counterexamples that have to be analyzed, which, in turn, is caused by the specifics of the formula and the program: Here, we are checking on a certain counter that counts the number of rising edges of an axis sensor; this must stay below 40. The brute force approach of the A_1 abstraction is better in this case.

4.7 CONCLUSION

In this chapter we introduced a CEGAR scheme specifically geared towards model checking PLC programs. We detailed different techniques that were used to abstract the state space while it is built (for input variables) and were guided by the analysis of spurious counterexamples (for local variables). Using these abstraction techniques we were able to verify various $\forall$CTL properties for programs and function blocks from academia and industry.

In the case study we could see that, in some cases, the brute force approach without the more complex organization of the state space could still be faster.

Obviously, the technique is limited by the power of the constraint solver. In the next section, we will, therefore, introduce a predicate abstract using existing SMT solvers. Finally, in Chap. 9, we will use static analysis results to analyze a complete safety application.

5

PREDICATE ABSTRACTION

The previous chapter introduced a CEGAR-based approach to iteratively build and check an abstract model for PLC programs. Yet, this approach is limited to the domains we have implemented in Arcade.PLC and the restrictions of the handwritten constraint solver. More complicated programs necessitate a more powerful approach. In this chapter, we detail a predicate abstraction [64] for PLC programs that abstracts the program behavior using predicates between variables. The actual transition relation, i. e., which predicates are fulfilled at which program locations, is then discovered using automatic decision procedures.

5.1 Overview & Outline

In the approach described in this section, we first encode the semantics of a given PLC program as first order logic formulae. That is, the instructions of our intermediate representation are formulated in a logic suitable for automatic solvers. We describe this encoding in Sect. 5.4.

In Sect. 5.5, we then automatically derive the transition relation of an abstracted state space. The abstraction is based on predicate expression over program variables. For each program location, we track the evaluation of these predicates using SMT solving. Additionally, we introduce a predicate scoping, which attaches a lifetime to certain predicates such that it is no longer necessary to keep their evaluation at each program location. This technique allows to further reduce the size of the state space.

The feasibility of our approach is demonstrated in Sect. 5.6 by checking various PLC programs. The chapter ends with a conclusion in Sect. 5.7. We start by discussing related work and then motivating our approach using a worked example, which is used throughout this chapter.

5.2 Related Work

In their seminal paper, Graf and Saïdi [64] showed how to derive abstract state spaces using decision procedures. Their approach works by adding all derived transitions as blocking clauses until a formula becomes unsatisfiable. Numerous

```
1  PROGRAM Example
2  VAR_INPUT
3    in0, in1, in2: USINT;
4    flag : BOOL;
5  END_VAR
6  VAR_OUTPUT
7    out : USINT;
8  END_VAR
9  VAR
10   var : USINT;
11 END_VAR
12 IF flag THEN
13   IF in0+in1+in2 < 100 THEN
14     var := in0;
15   ELSE
16     var := 0;
17   END_IF;
18 ELSE
19   out := var;
20 END_IF;
21
22 END_PROGRAM
```

Figure 12: Example PLC program used throughout this chapter [22]

works refined this approach in different directions. Ball et al. [7], e. g., make use of abstraction interpretation [44] for C code verification so as to derive the successors of multiple (unrelated) predicates in one decision procedure call. This also allows the representation of *don't cares* for the predicate evaluation. Henzinger et al. [69], on the other hand, introduce a lazy abstraction scheme, which works by using a different precision for different parts of the program and is deeply ingrained in their refinement loop. Our predicate scoping technique can be seen as a special case of these approaches, tailored for the cyclic scanning mode of PLCs.

5.3 WORKED EXAMPLE

Our approach is motivated with the small example program shown in Fig. 12, which is written in ST. The program performs the following operation in each cycle: First, the input variable `flag` is tested (line 12). If the flag is set, the program tests whether the sum of the three inputs `in0`, `in1`, `in2` is less than 100 (line 13). If so, the variable `in0` is copied into `var`, otherwise the variable `var` is set to 0. If `flag` is not set then the value of `var` is copied into the output variable `out` (line 19). Note that `var` is a non-temporary variable, which holds its value for the next cycle.

Verifying an Invariant

Suppose we want to manually verify that the invariant `out` < 100 holds for this program. We first observe that the variable `out` is only set to the value of `var`, which in turn is either set to 0 or `in0`. In the first case the invariant is trivially true. The second case can only be executed if $\texttt{in0} + \texttt{in1} + \texttt{in2} < 100$, which implies that $\texttt{in0} < 100$ (overflow cannot occur here, since arithmetic is implicitly cast to a bigger accumulator data type here), making the invariant true. Note that it is not obvious how to automate these steps to prove the invariant.

Unfortunately, the techniques we presented the previous chapter cannot readily be applied to the example. The reason for this is that the inequality $\texttt{in0} + \texttt{in1} + \texttt{in2} < 100$ cannot be abstracted efficiently by intervals, resulting again in a state explosion. One way to capture such expressions that relate the values between variables is to use more powerful domains such as convex polyhedra [46], difference bound matrices [80, 124], or octagons [87].

In this chapter, however, we extend our model checking in a different direction. Instead of implementing a new domain for this specific setup, we implement an abstraction over predicates. These predicates can be arbitrary Boolean expressions and are later evaluated using automatic decision procedures. In case of the example program, we would evaluate the predicate $\pi_0 := \texttt{in0} + \texttt{in1} + \texttt{in2} < 100$ (and other suitable predicates) at every program location. The state space itself then comprises of states that are tuples of the current line number and evaluation of the predicates or conjunctions of predicates. Note that this notation is different from our previous convention: We now evaluate intermediate steps, which are not observable, and thus should be hidden to the model checker. We hence have to slightly extend our specification of the invariant as follows:

Predicates

$$AG\,(\mathsf{exitpoint} \implies \texttt{out} < 100) \tag{1}$$

Here, exitpoint is an atomic proposition that evaluates to true only at the exit point of the program. This ensures that only the observable behavior of the program is verified and that non-observable intermediate states are hidden.

Exit Point

To automate the building of the state space, we will encode the program semantics in first order logic (FOL). Each statement will then relate preconditions to postconditions, and we can use solver calls to evaluate the validity of predicates in the program.

First Order Logic

5.4 ENCODING OF PLC SEMANTICS IN FOL

In this section, we describe the transformation of PLC programs into FOL formulae. Each model of such a formula represents a possible state change by the statement encoded in the formula. Syntactically, we use unprimed variables for the precondition and primed variables for the postcondition when encoding the program semantics. This allows to derive the transition relation between states with decision procedures. We start by encoding key components of PLC programs.

Precondition: x
Postcondition: x'

5.4.1 *Encoding of Variables and the Program*

As in Def. 2.2, let $\textsc{Var}$ be the set of variables of the PLC program. In this section, however, we flatten all structures and arrays so that we only deal with scalar variables. Indirect array accesses and pointers are not supported throughout this chapter. Depending on their lifetime and their semantics, we partition $\textsc{Var}$ into three distinct sets $\textsc{Var}_M$, $\textsc{Var}_I$ and $\textsc{Var}_T$ (variables that retain their value between cycles, inputs variables, and temporary variables, cp. Def. 2.2). Since recursion is not possible in PLC programs, we can determine the number of variables used such that each variable has a unique identifier/address. Special handling of local variables stored on a stack for function block or function calls is thus not required.

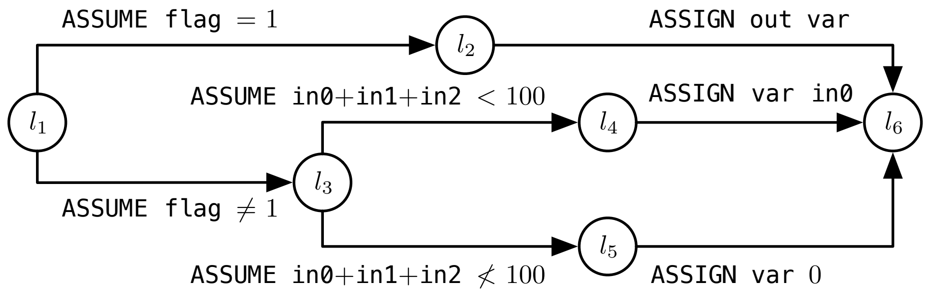


Figure 13: Control flow automaton of program `Example` [22].

Domain of Discourse

It the following, let $\mathcal{D}$ be our domain of discourse. The domain of discourse can be chosen as the union of the data types of the variables in $\textsc{Var}$, after flattening are all elementary (i. e., non-aggregate) data types. We introduce memory states, defined for each program location:

Memory State

Definition 5.1: A *memory state* is given by a tuple $\langle \ell, \nu \rangle$, where $\ell \in L$ is a program location and $\nu\colon \textsc{Var} \to \mathcal{D}$ is a variable assignment. Here, ℓ stands for a symbolic address (e. g., a line number) of the next statement to be executed.

We define two special program locations ℓ_S and ℓ_E which denote the entry point and exit point of the program[1]. The program model, which contains all possible executions of the PLC program, is defined in the following form:

Program Model

Definition 5.2: The *program model* is a state transition system $\langle S, I, R \rangle$ where S is the set of memory states, $I \subseteq S$ is the set of initial memory states and $R \subseteq S \times S$ is a transition relation.

Worked Example

Consider the worked example `Example` of Figure 12. Its set of variables is encoded as $\mathsf{Var} = \{\mathtt{in0}, \mathtt{in1}, \mathtt{in2}, \mathtt{flag}, \mathtt{out}, \mathtt{var}\}$, which is subdivided into $\textsc{Var}_I = \{\mathtt{in0}, \mathtt{in1}, \mathtt{in2}, \mathtt{flag}\}$, $\textsc{Var}_M = \{\mathtt{var}, \mathtt{out}\}$ and $\textsc{Var}_T = \{\}$. The domain of discourse is the union of the data types $\mathtt{BOOL} \cup \mathtt{USINT} \cup \mathtt{UDINT}$. The program model of the program would contain, e. g., the transition $(19, \langle \mathtt{out} = 1, \mathtt{var} = 0, \dots \rangle) \rightsquigarrow (20, \langle \mathtt{out} = 0, \mathtt{var} = 0, \dots \rangle)$.

The encoding described thus far contains even more states than the concrete model described in Def. 2.4 since we now consider intermediate states. To actually reduce the number of states, we will introduce a symbolic encoding using FOL formulae.

1 The program can always be transformed to have only one exit point.

5.4.2 *Translating PLC Programs as FOL Formulae*

As described in Sect. 3.4.4, we first compile all PLC programs into our intermediate representation (IR). In this chapter, we translate the IR into FOL in two steps. First, we translate the IR statements into a *control flow automaton* [12]. Then, the automaton is converted into FOL formulae.

Definition 5.3: A *control flow automaton* (CFA) is a labeled state transition system $\langle L, \textsc{Stmt}, G \rangle$ where L is a set of program locations, $\textsc{Stmt}$ is a set of operations over the variables and $G \subseteq L \times \textsc{Stmt} \times L$ is a set of control flow edges. *Control Flow Automaton*

With $\ell \in L$ we represent a location in the IR of the program. A control flow edge $\langle \ell, \cdot, \ell' \rangle$ indicates that if the current state is in location ℓ, then there is a possible transition to location ℓ' after the execution of the statement. We define two instructions that can be used as operations in the CFA:

- `ASSUME` π
- `ASSIGN` v t

The *assign* operation directly corresponds to all assignments used in our IR and the *assume* operation is used to model the guards of conditional jumps. The operation *assume* is followed by a predicate $\pi = p(t_1, \ldots, t_n)$ or a negated predicate $\pi = \neg p(t_1, \ldots, t_n)$, meaning that the transition is executed only if the given predicate application is valid. The operation *assign* is followed by a variable v and a term t of the same type. It means that in the next state the variable v will be assigned with the evaluation of the expression t in the current state. Here, a term t is defined as follows:

Definition 5.4: A *term* t is either a variable or a function application with other terms: *Term*

$$\begin{array}{rcl} t & ::= & x \\ & | & f(t_1, \ldots, t_n) \end{array}$$

where $x \in \textsc{Var}$ is a variable and f is a function symbol.

Note that constants are also defined as functions (with arity 0). We define all operators allowed in our IR (cp. Sect. 3.4.4) as functions, including the data type casts to different scalar types but without any pointer operations.

Consider again the program `Example` of Fig. 12. The labeled state transition system shown in Fig. 13 is the translation of the example program into a control flow automaton according to Def. 5.3. For accessibility of the presentation, we represent binary predicate and function applications using infix notation, which is interpreted according to the standard operator precedence, e. g., $\texttt{in0} + \texttt{in1} + \texttt{in2} < 100$ stands for $<(+(+(\texttt{in0}, \texttt{in1}), \texttt{in2}), 100)$. *Example*

Since we want to use SMT solving techniques to query the validity of predicates in the CFA, we translate the memory operations into quantifier-free FOL formulae that encode the assumptions and transformations over the memory. Quantifier free FOL formulae can be expressed in terms of predicate applications, negations and conjunctions.

Quantifier Free Formula

Definition 5.5: A quantifier free formula φ is inductively defined as follows:

$$\begin{array}{rcl} \varphi & ::= & p(t_1, \ldots, t_n) \\ & | & \neg\varphi_1 \\ & | & \varphi_1 \wedge \varphi_2 \end{array}$$

where φ_1, φ_2 are quantifier free formulae, p is a predicate symbol and $t_1, \ldots, t_n$ are terms according to Def. 5.4.

Next, we will describe the encoding of CFA operations into FOL formulae. Each operation is encoded as a relation of pre- and post-states of the transition. Pre- and post-variables are syntactically denoted by the set of unprimed and primed symbols.

Encoder enc

We define an encoder function $\mathsf{enc}\colon \textsc{Stmt} \to \mathcal{L}$, where $\mathcal{L}$ is the language of FOL formulae. The encoding of an *assume* operation is given by the assertion of the predicate on the pre-variables conjoined with the equality between all pre- and post-variables, which remain unchanged:

$$\begin{array}{lcr} \mathsf{enc}(\mathtt{ASSUME}\ p(t_1, \ldots, t_n)) & := & p(t_1, \ldots, t_n) \wedge \bigwedge_{y \in \textsc{Var}} y' = y \\ \mathsf{enc}(\mathtt{ASSUME}\ \neg p(t_1, \ldots, t_n)) & := & \neg p(t_1, \ldots, t_n) \wedge \bigwedge_{y \in \textsc{Var}} y' = y \end{array}$$

An assignment is encoded by asserting the equality between the post-variable that has to be assigned with the term over the pre-variables, conjunct with equalities between pre- and post- versions of the variables that remain unchanged:

$$\mathsf{enc}(\mathtt{ASSIGN}\ x\ t) \quad := \quad x' = t \wedge \bigwedge_{y \in \textsc{Var} \setminus \{x\}} y' = y$$

Using the encoder enc, we can now derive a symbolic encoding of the CFA:

Symbolic Encoding

Definition 5.6: A control-flow-based symbolic encoding of a PLC program with CFA $\langle L, \textsc{Stmt}, G\rangle$ using FOL formulae is then given by the following:

- The variables and their domain $\langle \textsc{Var}, \mathcal{D}\rangle$,
- the transition system $\langle L, \mathcal{L}, G\rangle$, where all operations in Stmt of the CFA are encoded as FOL formulae in $\mathcal{L}$ using the encoder enc,
- the start-up phase $Start \in L \times \mathcal{L}$, and
- the scanning phase $Scan \in L \times \mathcal{L}$.

Start-up Phase

In this definition, the *start-up phase* defines the initialization of all variables in the initial location ℓ_S:

$$Start := \left\langle \ell_S, \bigwedge_{x \in \textsc{Var} \setminus \textsc{Var}_I} x = \mathsf{Init}_x() \right\rangle,$$

Scanning Phase

where $\mathsf{Init}_x()$ is the default value x is initialized to. The *scanning phase* captures the behavior of the controller between cycles, i. e., after the execution of the program body has reached the last program location $\ell_E \in L$. During this phase, the values of the variables in $\textsc{Var}_M$ are retained while input variables are read from the environment; hence, their value becomes non-deterministic:

$$Scan := \left\langle \ell_E, \bigwedge_{x \in \textsc{Var} \setminus \textsc{Var}_I} x' = x \right\rangle$$

We can now define a symbolic encoding of a program as a tuple $\langle S, I, R \rangle$ as in Def. 5.2, which can be handled with SMT solving techniques [10]. Given a theory $\mathcal{T}$ chosen for the interpretation of variables and predicates, the set of states S is defined as the set of all locations and all possible assignments consistent under $\mathcal{T}$:

$$S := \{\langle \ell, \nu \rangle \mid \ell \in L \text{ and } \nu \in \textsc{Var} \to \mathcal{D}\}$$

The set of initial states is defined as all those states in the start-up location with all variables initialized accordingly:

$$I := \{\langle \ell_S, \nu \rangle \mid \nu \models_{\mathcal{T}} \varphi \text{ where } \langle \ell_S, \varphi \rangle = Start\}$$

The transition relation is given by pairs of states. Each transitions covers consecutive locations such that the assignment of the first state over unprimed variables ν_1 and the assignment of the second state over primed variables ν_2' satisfy the formula. Additionally, a transition is possible from the last location to the first location using the scanning phase:

$$\begin{aligned} R := &\left\{ \langle \langle \ell, \nu_1 \rangle, \langle \ell', \nu_2 \rangle \rangle \mid \nu_1, \nu_2' \models_{\mathcal{T}} \varphi \text{ and } \langle \ell, \varphi, \ell' \rangle \in G \right\} \;\cup \\ &\left\{ \langle \langle \ell_E, \nu_1 \rangle, \langle \ell_S, \nu_2 \rangle \rangle \mid \nu_1, \nu_2' \models_{\mathcal{T}} \varphi \text{ and } Scan = \langle \ell_E, \varphi \rangle \text{ and } Start = \langle \ell_S, \cdot \rangle \right\}. \end{aligned}$$

5.4.3 *Encoding of Timers*

PLC programs can react to timer events using the standard timer FBs `TP`, `TON` and `TOF`, which we described in Sect. 2.2.6. To support these timers, we extend the set of statements $\textsc{Stmt}$ with the following operations:

- `TP` $n\ t_{\texttt{IN}}\ t_{\texttt{PT}}$,
- `TON` $n\ t_{\texttt{IN}}\ t_{\texttt{PT}}$,
- `TOF` $n\ t_{\texttt{IN}}\ t_{\texttt{PT}}$,

where $n \in \textsc{Timer}$ is the name of the timer and $t_{\mathtt{IN}}$ (timer input) and $t_{\mathtt{PT}}$ (programmed time) are terms. We assume that for each timer $n \in \textsc{Timer}$ the input and output variables $n.\mathtt{IN}$ (timer input), $n.\mathtt{Q}$ (timer output) are part of $\textsc{Var}$. For timers of type TON and TOF the propositional variable $n.r$ is added to $\textsc{Var}$, which keeps track of whether the timer is running.

Timer TP starts (setting Q to 1) if Q is 0 and there is a rising edge on input IN; nothing is changed otherwise:

$$\begin{aligned}\mathsf{enc}(\mathtt{TP}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}) \quad := \quad & n.\mathtt{IN}' = t_{\mathtt{IN}} \wedge \textstyle\bigwedge_{x \in \textsc{Var} \setminus \{n.\mathtt{IN}, n.\mathtt{Q}\}} x' = x && \wedge \\ & ((n.\mathtt{Q} = 0 \wedge n.\mathtt{IN}' > n.\mathtt{IN}) \rightarrow n.\mathtt{Q}' = 1) && \wedge \\ & (\neg(n.\mathtt{Q} = 0 \wedge n.\mathtt{IN}' > n.\mathtt{IN}) \rightarrow n.\mathtt{Q}' = n.\mathtt{Q})\end{aligned}$$

Timer TON is started on a rising edge of IN and stops (setting Q to 0) on falling edges. It is defined as follows:

$$\begin{aligned}\mathsf{enc}(\mathtt{TON}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}) \quad := \quad & n.\mathtt{IN}' = t_{\mathtt{IN}} \wedge \textstyle\bigwedge_{x \in \textsc{Var} \setminus \{n.\mathtt{IN}, n.\mathtt{Q}, n.r\}} x' = x && \wedge \\ & (n.\mathtt{IN}' > n.\mathtt{IN} \rightarrow n.r' \wedge n.\mathtt{Q}' = n.\mathtt{Q}) && \wedge \\ & (n.\mathtt{IN}' < n.\mathtt{IN} \rightarrow \neg n.r' \wedge n.\mathtt{Q}' = 0) && \wedge \\ & (n.\mathtt{IN}' = n.\mathtt{IN} \rightarrow n.r' \leftrightarrow n.r \wedge n.\mathtt{Q}' = n.\mathtt{Q})\end{aligned}$$

Timer TOF is started on a falling edge of IN and stops (setting Q to 1) on rising edges. It is defined as follows:

$$\begin{aligned}\mathsf{enc}(\mathtt{TOF}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}) \quad := \quad & n.\mathtt{IN}' = t_{\mathtt{IN}} \wedge \textstyle\bigwedge_{x \in \textsc{Var} \setminus \{n.\mathtt{IN}, n.\mathtt{Q}, n.r\}} x' = x && \wedge \\ & (n.\mathtt{IN}' > n.\mathtt{IN} \rightarrow \neg n.r' \wedge n.\mathtt{Q}' = 1) && \wedge \\ & (n.\mathtt{IN}' < n.\mathtt{IN} \rightarrow n.r' \wedge n.\mathtt{Q}' = n.\mathtt{Q}) && \wedge \\ & (n.\mathtt{IN}' = n.\mathtt{IN} \rightarrow n.r' \leftrightarrow n.r \wedge n.\mathtt{Q}' = n.\mathtt{Q})\end{aligned}$$

We extend the start-up phase by disabling all timers and setting all variables to 0. The scanning phase is extended by the encoding of the elapsing time $\mathcal{TE}_t(n)$ for each timer n of type t: Timer TP can have a falling edge on Q if running, where as it remains disabled with Q set to 0 otherwise:

$$\mathcal{TE}_{\mathtt{TP}}(n) := n.\mathtt{IN}' = n.\mathtt{IN} \wedge n.\mathtt{Q} = 0 \rightarrow n.\mathtt{Q}' = 0$$

Timer TON (TOF) either remains unchanged or can have a rising (falling) edge on Q if running:

$$\begin{aligned}\mathcal{TE}_{\mathtt{TON}}(n) &:= n.\mathtt{IN}' = n.\mathtt{IN} \wedge n.r' \leftrightarrow n.r \wedge (n.\mathtt{Q}' = n.\mathtt{Q} \vee n.r \wedge n.\mathtt{Q} = 0 \wedge n.\mathtt{Q}' = 1) \\ \mathcal{TE}_{\mathtt{TOF}}(n) &:= n.\mathtt{IN}' = n.\mathtt{IN} \wedge n.r' \leftrightarrow n.r \wedge (n.\mathtt{Q}' = n.\mathtt{Q} \vee n.r \wedge n.\mathtt{Q} = 1 \wedge n.\mathtt{Q}' = 0)\end{aligned}$$

5.4.4 *Succinct Representation of Control-Flow Automata*

In our definition, we used one transition in the CFA for each instruction of the IR. Such an encoding that handles one instruction per transition is called *single-block encoding* (SBE). It has the following drawbacks: First, we have to compute the evaluation of all formulae at each intermediate step, even if the intermediate results are not needed (or not needed in this precision) for further steps. Second, the conjunction of multiple intermediate steps (i. e., FOL formulae) might be easier and thus faster to evaluate using automated decision procedures than each step on its own. Such a conjunction of simple intermediate steps without change in control flow is called a *basic block encoding* (BBE). This idea can further be improved to control flow trees, which then is called *extended-basic-block encoding* (EBBE), and even loop free fragments, called *large block encoding* (LBE) [12].

In our approach, we use a BBE. This can simply be achieved by conjoining the formulae of all basic blocks. In this encoding, a basic block of the CFA is defined as control flow edges $\ell_0 \rightarrow \cdots \rightarrow \ell_i$, with

- ℓ_0 has exactly one successor and has more than one predecessors or is ℓ_S,
- $\ell_1, \ldots, \ell_{i-1}$ have exactly one predecessors and exactly one successor, and
- ℓ_i has exactly one predecessor and has more than one successor or is ℓ_E.

These basic blocks can easily determined in the CFA. We then conjoin their formulae and shrink them to a single transition.

5.5 PREDICATE ABSTRACTION

Let $P = \{\pi_1, \ldots, \pi_n\}$ be a set of predicates over the set of variables VAR, which we call the *abstraction precision*. The Boolean predicate abstraction of a system computes an over-approximation that keeps track where in the program each of the predicates in P is valid or not [64, 7].

Abstraction Precision

Definition 5.7: We define an *abstract state* as a tuple $\langle \ell, c \rangle$ of a location ℓ and a minterm c over a set $B = \{b_1, \ldots, b_n\}$ of Boolean variables. A minterm over B is a conjunction of all variables $b_i \in B$, where each b_i appears either with or without negation.

Abstract State

The intuition here is that each b_i corresponds to a predicate π_i:

Definition 5.8: The *abstraction function* α maps a memory state $\langle \ell, v \rangle$ to an abstract state $\langle \ell, c \rangle$, in which the polarity of each variable in c states the validity of the respective predicate in v:

Abstraction Function

$$\alpha(\langle \ell, v \rangle) := \langle \ell, c \rangle \text{ such that for all } 1 \leq i \leq n : c \models b_i \text{ iff } v \models_{\mathcal{T}} \pi_i$$

We define the concretization function as the inverse of the abstraction function $\alpha^{-1}(\hat{s}) := \{s \mid \hat{s} = \alpha(s)\}$. The abstraction function over-approximates, i. e., the abstraction $\alpha(s)$ of a state s represents a region $s \in \alpha^{-1}(\alpha(s))$ of states in which s is contained.

Since we are interested in verifying universal properties, we want the abstraction to be conservative for such properties. First, we guarantee this by assuring that all predicates of the property are contained in the precision. Second, since we over-approximate the system, i. e., $S \subseteq \alpha^{-1}(\alpha(S))$, it follows that if we prove the set of reachable states $S^{\rightarrow}$ to satisfy the property in the abstract system, then it is valid in the concrete system as well. In general, the vice-versa does not hold, which gives rise to CEGAR techniques [40], which were highlighted in Chap. 4.

5.5.1 *Implementation of the Predicate Abstraction*

The predicate abstraction allows us to represent a program $\langle S, I, R\rangle$ as a Boolean over-approximation in terms of a Kripke structure $\langle \hat{S}, \hat{I}, \hat{R}, \hat{AP}, \hat{L}\rangle$ (cp. Sect. 2.4.1 and 2.5.2) where

- $\hat{S} := \{\alpha(s) \mid s \in S\}$ is the set of abstract states,
- $\hat{I} := \{\alpha(s) \mid s \in I\} \subseteq \hat{S}$ is the set of initial states,
- $\hat{R} := \{\langle \alpha(s), \alpha(s')\rangle \mid \langle s, s'\rangle \in R\} \subseteq \hat{S} \times \hat{S}$ is the transition relation,
- $\hat{AP}$ is the set of atomic propositions, and
- $\hat{L}\colon \hat{S} \to 2^{\hat{AP}}$ is the labeling function defined as $\hat{L}(\langle \ell, c\rangle) := \{b \in \hat{AP} \mid c \models b\}$.

We generate this Kripke structure on-the-fly by providing two main functions: *precondition* and *strongest postcondition*. The precondition $p \subseteq \hat{S}$ represents the set of abstract initial states $\hat{I}$. It is defined as the set of abstract states $\langle \ell_S, c\rangle$ at start location, whose minterms are entailed by the start condition:

Precondition

$$p := \{\langle \ell_S, c\rangle \mid \langle \ell_S, \varphi\rangle = \mathit{Start} \text{ and } c \wedge \varphi \wedge \textstyle\bigwedge_{i=1}^{n}(b_i \leftrightarrow \pi_i) \text{ is } \mathcal{T}\text{-}\textsc{Sat}\}\,.$$

The strongest postcondition $sp\colon \hat{S} \to 2^{\hat{S}}$ represents to the set of successors of an abstract state under the abstract transition relation $\hat{R}$. Both $\hat{I}$ and $\hat{R}$ are sets of abstracted states. Given a fixed location, sp can hence be characterized as the enumeration of all minterms over B that are $\mathcal{T}$-satisfiable when conjoined with the set and the *abstraction constraint* $\bigwedge_{i=1}^{n}(b_i \leftrightarrow \pi_i)$. The strongest postcondition of an abstract state $\langle \ell, c_1\rangle$ can hence be defined as the set of abstract states $\langle \ell', c_2\rangle$ at successor locations such that the minterms c_1 and c_2 abstract the transition formula on the pre- and post-variables:

Strongest Postcondition

$$\begin{aligned} &sp(\langle \ell, c_1\rangle) := \\ &\quad \{\langle \ell', c_2\rangle \mid \langle \ell, \varphi, \ell'\rangle \in G \text{ and } c_1 \wedge c_2' \wedge \varphi \wedge \textstyle\bigwedge_{i=1}^{n}(b_i \leftrightarrow \pi_i \wedge b_i' \leftrightarrow \pi_i') \text{ is } \mathcal{T}\text{-}\textsc{Sat}\}\,. \end{aligned}$$

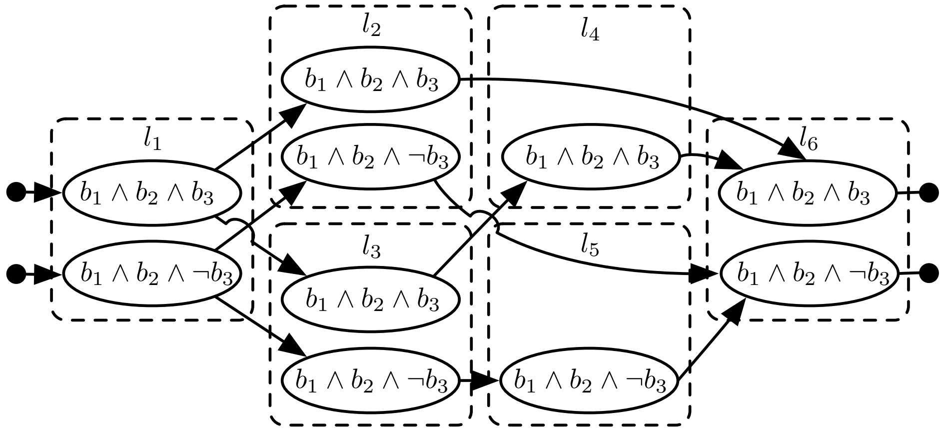


Figure 14: Predicate abstraction of `Example`. Solid circles on the left and right indicate the scanning phase and are connected by transitions [22].

Computing these equations can be seen as an *AllSAT* problem over a set of variables [78], which in our case are B for the precondition and B' for the strongest postcondition. In our implementation, we use the Z3 SMT solver [89] to compute this automatically. We iterate over each outgoing transition $\langle \ell, \varphi, \ell' \rangle \in G$ explicitly. For each location ℓ' we query the Z3 SMT solver for models of the formula. If it returns a model, we extract a minterm c_2', which corresponds to an abstract state $\langle \ell', c_2 \rangle$. Then we conjoin the blocking clause $\neg c_2'$ to the formula and repeat the process, calling Z3 again. When the formula becomes unsatisfiable, we have seen all models. When iterating among different outgoing transitions directed to the same location ℓ', we introduce blocking clauses to avoid double occurrences. We memoized the result so as to avoid unnecessary solver calls.

Example Continued

We continue our worked example from Sect. 5.3 verifying (1). We start with the set of predicates $P = \{\pi_1 = (\texttt{out} < 100)\}$, since the property to verify is always part of the precision. The property we want to verify thus becomes $AG\ b_1$[2]. Since we do not have any restriction on `var`, a counterexample is generated. At the end of the counterexample, the property $\texttt{var} \geq 100$ holds, which is then assigned to `out`. This gives rise to the new predicate $\pi_2 = (\texttt{var} < 100)$, which we add to P and rerun the process. In the next refinement step we similarly detect: $\texttt{in}_0$ is assigned to `var`, we hence deduce $\pi_{\text{skip}} = (\texttt{in}_0 < 100)$[3]. Finally, we discover that the previous statement can only be executed if the predicate $\pi_3 = (\texttt{in}_0 + \texttt{in}_1 + \texttt{in}_2 < 100)$ is satisfied (from ℓ_3 to ℓ_4), so π_3 is added to P. The abstracted state space using predicates of P is shown in Fig. 14, where each b_i represents the validity of the corresponding predicate π_i. The final state space allows us to verify formula (1), since b_1 is valid everywhere.

2 We ignore the exitpoint predicate of the program to make the presentation more accessible.
3 We skip this predicate, since it is not needed and clutters the presentation as well.

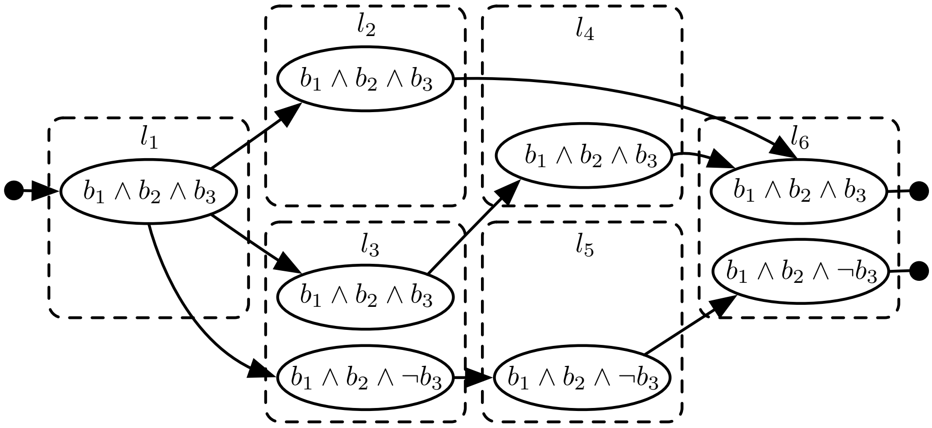


Figure 15: Predicate abstraction with b_3 scoped to $\langle \ell_3, \ell_6 \rangle$. Solid circles on the left and right indicate the scanning phase and are connected by transitions [22].

5.5.2 *Scoping of Predicates*

In the previous section, we have evaluated all predicates in every location, i. e., we have evaluated $b_1, \ldots, b_n$ to either true or false without the possibility to leave it as *unknown*. This approach is potentially wasteful: Consider the running example again with notation as in Fig. 14. For the initial location ℓ_1, we have to consider the two states $(b_1 \wedge b_2 \wedge b_3)$ and $(b_1 \wedge b_2 \wedge \neg b_3)$. Note that the predicate π_3 (and thus the evaluation b_3) is of no use in the initial state but only in ℓ_3. In particular, it also pollutes the path $\ell_1 \to \ell_2 \to \ell_6$, where it plays no role. In this section we will therefore reduce the scope of certain predicates and first define:

Weak Reachability

Definition 5.9: Let $\langle L, \cdot, G \rangle$ a control flow automaton. The *weak reachability* relation $\preceq \subseteq L \times L$ is defined as follows:

$$\ell \preceq \ell'' \text{ iff } \ell = \ell'' \text{ or there exists } \left\langle \ell, \cdot, \ell' \right\rangle \in G \text{ such that } \ell' \preceq \ell''.$$

In other words, two locations are weakly reachable if there is a path of locations between them.

Note that the weak reachability is a purely syntactic notation. The reachability does not consider the transition over data variables, hence two weakly reachable locations could be not actually reachable in a real execution. To each predicate π_i we now associate a scope $\left\langle \check{\ell}_i, \hat{\ell}_i \right\rangle \in L \times L$. We then redefine the abstraction function in a way that predicates are used only if they are in the given scope:

$$\alpha(\langle \ell, \nu \rangle) := \langle \ell, c \rangle \text{ such that for all } 1 \leq i \leq n : c \models b_i \text{ iff } (\check{\ell}_i \preceq \ell \preceq \hat{\ell}_i \implies \nu \models_{\mathcal{T}} \pi_i)$$

We use the weakest preconditions to automatically limit the scope for new predicates. If we have a sequence of consecutive preimages with common predicate

$\langle \ell_1, \varphi_1 \rangle, \ldots, \langle \ell_m, \varphi_m \rangle$, those predicates will use the scope $\langle \ell_1, \ell_m \rangle$. If this sequence passes through the scanning phase, we break it up into two different predicates with scope $\langle \ell_1, \ell_E \rangle$ and $\langle \ell_S, \ell_m \rangle$, respectively. If the sequence passes through the scanning phase more than once, we do not introduce a scoping.

Example Continued

In our example program we would now associate the predicate scope $\langle \ell_3, \ell_6 \rangle$ to π_3. This means that, e.g., in state ℓ_1 the variable b_3 is not evaluated and fixed to *true*. Hence only the state $b_1 \wedge b_2 \wedge b_3$ appears in ℓ_1, further reducing the number of states and transitions. The complete state space using this scoping is depicted in Fig. 15. Although the scoping is only able to reduce the number of states by two here, we will see how powerful the scoping is in the next section.

5.6 CASE STUDY

Our approach is implemented in the ARCADE.PLC framework. To show the effectiveness of the approach beyond the running example, we applied it to various FBs [22][4]. All experiments were performed on a MacBook Pro equipped with an Intel Core i5 processor with 2.53 GHz and 8 GB of main memory.

Setup

For the case study, we again selected two complex safety-critical FBs from the PLCOPEN library [94]. We used our own implementation of the PLCOPEN library implemented in ST in these experiments. The `SF_ModeSelector` FB has 14 inputs, 12 outputs and 5 internal variables and is implemented in 175 lines of ST. It controls that (up to eight) different modes of operation of a machinery are selected in a consistent way, i.e., that one mode, at most, is active at a time. Further, only for a short period of time (while switching modes) it is possible that no mode is selected. It additionally allows for locking of modes. We verified that (1) one mode, at most, is selected at a time, and that (2) exactly one mode is selected if it is locked.

Moreover, we verified the `SF_MutingPar` FB which allows for muting a safety function while monitoring that certain safety sensors are operated in the correct order. It has 13 inputs and 12 internal variables. We first verified that the FB only signals *Ready* when it is activated (3). Afterwards we tried to verify that a certain safety output (AOPD) is only set when the muting lamp is switched on (4).

Evaluation

The results are shown in Tab. 4. The columns of the table indicate in order: The program, the formula checked, the abstraction used ("—" = abstractions from the previous chapter, PA = predicate abstraction, PS = predicate abstraction with predicate scoping), the number of states in the model, the number of transitions, the number of predicates used, the time for generating the abstract state space and model checking (where OOM means out of memory) and the total runtime (including predicate discovery).

The model checking of the successful examples only took seconds. In one example, finding the predicates was slow and took 70 s. The predicate scoping reduces

4 In this previous work [22], we also verified a safety property of a safety application. We later found a bug in the implementation of this application, which kept the safety output stuck to zero. This made the verification trivial. We, therefore, do not report this application here.

Program	φ	Abs.	#loc	#states	#trans.	#P	t_{abs}	t_{total}
`Example`	(1)	—	22	> 4k	> 40M	n/a	OOM	OOM
`Example`	(1)	PA	22	40	19	4	1 s	1 s
`Example`	(1)	PS	22	10	13	5	1 s	1 s
`ModeSelector`	(1)	PS	190	95	142	1	1 s	1 s
`ModeSelector`	(2)	PA	190	> 27k	> 28k	> 40	OOM	OOM
`ModeSelector`	(2)	PS	190	214	291	30	2 s	72 s
`MutingSeq`	(3)	PS	211	241	374	1	3 s	1 s
`MutingSeq`	(4)	PS	211	> 10k	> 14k	> 100	OOM	OOM

Table 4: Evaluation of the predicate abstraction and predicate scoping techniques [22]

the state space further: We were not able to verify formula (2) without predicate scoping. This example also shows the force of this abstraction: Although 30 predicates were in use, the final state space comprised only 214 states. The muting FB shows that sometimes simple invariants can be proven using a single predicate as in (3). Yet, our approach still not scales well enough to prove (4).

Regarding the runtime, we can observe that the actual model-checking process is performed in seconds even for the most complex programs. If the initial abstraction is not sufficient, refinement steps are necessary, which can be quite costly as shown with formula (2) where this takes 70 s of the total runtime. This predicate discovery seems to be the limiting factor of our current approach.

5.7 CONCLUSION

In this chapter, we introduced a fully automatic predicate abstraction for PLC programs. The abstraction works by first translating our intermediate representation of the PLC program into a control flow automaton, with edges represented using first order logic. Then, repeated SMT solver calls are used to discover the validity of predicates in the automaton. New predicates are automatically derived using counterexample analysis. To compute less predicates in fewer locations and thus reduce the size of the generated state spaces, we limit the scope of predicates to program locations where they actually influence the program semantics.

The technique, in its current form, it is not always better than the approach described in Chap. 4. The power of the predicate abstraction comes into play once more complex predicates are necessary, e. g., predicates that relate variables using arithmetic expressions, which is not always the case for the programs we checked. In this case, the brute force approach using intervals can be faster than expensive SMT solver calls. As the example programs shows, however, once more complex predicates are required, the approach shown in Chap. 4 can no longer compete.

6

MODEL CHECKING USING SAFETY AUTOMATA SPECIFICATIONS

Thus far, model checking of CTL formulae was used to verify safety-critical properties of PLC programs and function blocks. During our case studies, we faced two disadvantages in the usability of this technique:

- The formalization of properties in CTL turned out to be cumbersome and tedious. Complex properties were often wrong in the first attempt. Hence, we focused mostly on proving invariants of the program.
- Due to practical considerations, only a subset of the properties of the function blocks could be translated into CTL. While this was sufficient to verify the crucial safety-critical properties of a block, subtle problems in implementation details might have been missed.

In this chapter, we introduce *safety automata* as another specification formalism that addresses both problems: Specifications written as safety automata are easier and more intuitive to write than CTL and—in many cases—the complete specification can be proven. We show how this formalism can be used in the verification process of PLC programs and function blocks.

6.1 MOTIVATION & OVERVIEW

In Chap. 4 and Chap. 5 we investigated abstraction techniques to make the verification of PLC function blocks and programs feasible. During the verification, different properties had to be translated into CTL. For the verification of PLCopen function blocks, e. g., we first considered the textual description of the blocks that contains the most important (and safety-critical) properties of the block. Additionally, the description using state diagrams gave rise to further properties. Ideally, however, the whole specification of a function block should be verified in a more direct way. During the experiments with CTL, the author came to a similar conclusion about the usability of CTL as Schlipf et al. [107, p. 5]:

> *We found only simple CTL equations to be comprehensible; nontrivial equations are hard to understand and prone to error.*

Hence, we only verified selected properties, which is an unsatisfactory approach when trying to prove the correctness of the whole program. To make the formalization of safety specifications more intuitive, different solutions were proposed in the past:

- Pattern based approaches [31, 74, 52] allow the user to select a desired property from a list of predefined pattern. While this is a convenient and user-friendly approach, such pattern are either inflexible or tend to explode combinatorially [57, slide 12].
- Standardized subsets of natural languages allow the user to specify the properties using plain English or German. As an example, *Sicherheitsfachsprache* as defined by Mertke [84] allows for writing specifications in a subset of the German language.

By way of contrast, our approach is based on the existing formalism in which PLCOPEN function blocks are specified. This formalism is called *state diagram*; an example is shown in Fig. 18 on p. 83. While these automata are used by PLCOPEN to define the operation of the function blocks, we interpret them as safety automata similar to [55] and use them for verification.

6.1.1 *Bibliographic Notes & Related Work*

A multitude of alternative graphical specification paradigms have been researched in the past. Dillon et al. [51] propose the Graphical Interval Logic, which gives the user an intuitive graphical view on the specified property. Damm and Harel [48] propose *Live Sequence Charts* (LSC) as a visual formalism. LSCs build on Message Sequence Charts, a formalism to depict the interaction between processes or objects (often in a networked environment), but add temporal operators to it. Autili et al. [3] advocate *Property Sequence Charts* (PSC) another graphical formalism to specify the order of events. Finally, Asteasuain and Braberman [2] introduce *Featherweight Visual Scenarios*, a graphical, event-based specification language.

Our approach, however, is automata-based. It is close to *PLC-automata* [50] proposed by Henning Dierks as a formalism for the specification and verification of real-time systems. PLC-automata can be seen as a subset of timed automata and allow, in contrast to our approach, for specifying timing constraints. Our safety automata are inspired by an industry standard [94]. They are simple to verify and additionally provide the possibility to detect over-specifications, which we successfully used to detect an error in an industrial specification.

The safety automata presented in this chapter were initially developed during a masters thesis [34] and introduced to ARCADE.PLC as alternative formalism for model checking microcontroller code as well as PLC code. They are currently evaluated from the user's perspective to verify automotive software in an on-going thesis.

6.1.2 *Contribution & Outline*

First, we will formally define safety automata and some extensions in Sect. 6.2, and describe their relation to CTL. Then, we describe a model checking algorithm for safety automata in Sect. 6.3 that also handles certain extensions defined in this chapter. This algorithm is implemented in ARCADE.PLC and used in Sect. 6.4 to check an industrial PLCOPEN library. During the case study, we found a mistake in a specification. We detected this mistake, since a transition of an automata was never taken during model checking. The details how such erroneous specifications can be detected automatically are explained in Sect. 6.5. The chapter ends with a concluding discussion in Sect. 6.6.

6.2 SAFETY AUTOMATA

The key idea of safety automata is that they are designed to recognize all *safe* behavior of a program. An execution trace that cannot be recognized by the automaton will be signaled as an error. In principle, safety automata are non-deterministic finite automata:

Safety Automaton

Definition 6.1: Let AP be a set of atomic propositions. A safety automaton is a non-deterministic finite automaton defined as a tuple $\mathcal{A} = (Q, q_0, \Sigma, \delta, \mathcal{I})$, comprising a set of *states* Q, an initial state $q_0 \in Q$ and a transition function $\delta \subseteq Q \times \Sigma \times Q$ between states, where $\Sigma = 2^{\text{AP}}$. The *invariant* map $\mathcal{I} : Q \to \Phi_{\text{AP}}$ labels each state $q \in Q$ with an invariant $\mathcal{I}(q)$ given in propositional logic over AP.

The key idea here is that a safety automaton recognizes traces of the program. Therefore, each transition $(q, G, q') \in \delta$ is labelled with a guard $G \in \Sigma$. If the proposition of the guard is fulfilled, the corresponding transition has to be taken. If there are multiple transitions with a valid guard, a non-deterministic choice is taken. Usually, the guards specify certain conditions on the program inputs, possible error conditions, the firing of timers, etc. The states of the automaton are labeled with invariants over AP. These invariants specify, e. g., a behavior of the outputs or the local variables of the program. The automaton can remain in a state as long as the invariant is fulfilled. A trace of the program is then recognized by the automaton if and only if it can be recognized under these rules.

6.2.1 *Formalization*

To formalize, let $\pi = \langle s_0, s_1, \ldots, s_n \rangle$ be a trace of the PLC program. That is, s_0 corresponds to the initial state, and s_i is a configuration of the PLC after the execution of one cycle beginning at configuration s_{i-1}. By $s \models I$ we denote that a state fulfills an invariant $I \in \Phi_{\text{AP}}$, and by $s \models G$ that it fulfills a guard $G \in \Sigma$. Then, a trace π is recognized by the safety automaton $\mathcal{A}$, written $\pi \in L(\mathcal{A})$, iff: There exists $\langle q_0, \ldots, q_n \rangle \in Q^{n+1}$, $\langle G_1, \ldots, G_n \rangle \in \Sigma^n$ such that:

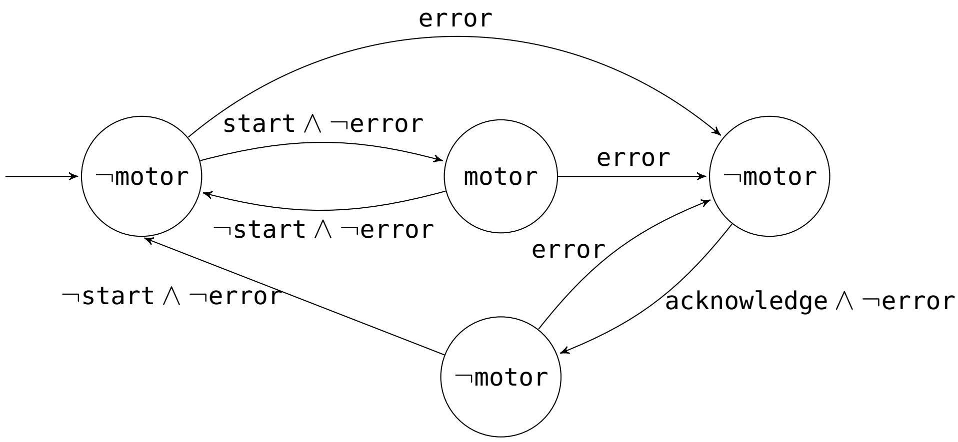


Figure 16: Example for a safety automaton.

1. $s_i \models \mathcal{I}(q_i)$ for all $0 \leq i \leq n+1$, and
2. $s_i \models G_i$ and $(q_{i-1}, G_i, q_i) \in \delta$ for all $1 \leq i \leq n$.

Observe that there is no guard for the first state, only an invariant. Every successor state then has one guard (from the transition) and one invariant (from the target state).

Acceptance Criteriton Finally, we can generalize the acceptance of safety automata to models of PLC programs: A model $(\mathcal{M}, s)$ is accepted by $\mathcal{A}$ iff for every path π (starting in the initial state s of the state space) we have $\pi \in L(\mathcal{A})$.

6.2.2 *Simplifications & Conventions*

To make safety automata more succinct, we introduce some conventions. Firstly, we assume an implicit back-edge at each node. This back-edge allows the automaton to remain in its current state when no guard is fulfilled. Formally, let G_i^* be the conjunction of all guards for all outgoing transitions in state $q_i \in Q$, i.e.:

$$G_i^* := \bigwedge \{G_j \mid \exists q_k : (q_i, G_j, q_k) \in \delta\} \tag{1}$$

We then always assume that $(q_i, \neg G_i^*, q_i) \in \delta$. This ensures that in each state at least one guard is always fulfilled. If this back-edge is not explicitly given, we implicitly assume the existence of this edge.

Example To exemplify, a simple safety automaton is given in Fig. 16. This automaton monitors a `motor`, `start`, `error` and `acknowledge` variable. In this example, `start`, `error` and `acknowledge` are inputs and `motor` is an output. The automaton then monitors that the motor is only activated as long as `start` is activated, and, if an error occurs, the motor is stopped and can only be restarted after `acknowledge` has been set. Note that in each state, only the relevant inputs are listed. According to our simplification, we can omit all changes to inputs that do not influence transitions between states. In the first state, e. g., the value of `acknowledge` does not influence any transitions and thus was omitted in the example.

6.2.3 *Relation to* CTL

The expressiveness of safety automata compared to CTL is given as follows:

Expressiveness

Proposition 6.1: Neither the set of languages described by safety automata $\mathcal{L}_{\mathsf{SA}}$ nor by $\mathcal{L}_{\mathsf{CTL}}$ are proper subsets of each other, i. e., some properties can only be described by safety automata, while others can only be described by CTL.

Proof. We give examples for properties the are recognized by one but not the other formalism.

$\mathcal{L}_{\mathsf{SA}} \subsetneq \mathcal{L}_{\mathsf{CTL}}$: CTL allows to express liveness properties such as $AG\ AF\ \varphi$, which is not possible using safety automata, since they only allow for specifying safety properties.

$\mathcal{L}_{\mathsf{CTL}} \subsetneq \mathcal{L}_{\mathsf{SA}}$: Safety automata are able to modulo-count events. That is, by creating a loop in the specification of the required length they can, e. g., test whether an odd number of events occurred. This is not possible in CTL, but requires extensions [79].

□

6.3 A MODEL CHECKING ALGORITHM FOR SAFETY AUTOMATA

A model checker for a safety automaton $\mathcal{A}$ has to check whether each trace of the Kripke structure $\mathcal{M}$ is in $L(\mathcal{A})$, i. e., accepted by $\mathcal{A}$. For this, we developed an on-the-fly checking algorithm, that builds the (abstracted) state space while checking whether the state space is accepted by $\mathcal{A}$. If not, a counterexample can be generated, which explains why the state space is not accepted by the automaton.

6.3.1 *On-the-fly Checking*

Algorithm 1 shows the on-the-fly checking algorithm. As input, the algorithm takes an initial state s_0 (from which the state space is built on the fly), the safety automaton $\mathcal{A}$ and, optionally, an *AcceptState* or *AlarmStates* (cp. Sect. 6.3.3).

The key idea of this algorithm is that it explores the synchronous product of the state space and the safety automaton. For this, a worklist queue is maintained that contains the tuples to be explored. Each tuple comprises (a) a state s from the state space and (b) a set $Q' \subseteq Q$, which reflects the possible states the safety automaton can assume when reaching state s from s_0. In each step, a tuple $\langle s, Q' \rangle$ from the worklist is processed. For each successor state s' of s all possible transitions of the safety automaton are determined. For this step, it is checked which guards and which invariants are satisfied. If there is an s', such that the automaton admits no possible transitions for no $q' \in Q'$, then a counterexample is found and returned (line 26). Otherwise, successor tuples that have not been visited yet are put on the worklist. If the worklist becomes empty we have visited each state s of the state space and now know that each path ending in s has at least one successor state admissible by the automaton. Hence, the state space is accepted.

6.3.2 *Counterexamples*

During model checking, our algorithm keeps track of the possible states reachable in the safety automaton while building the state space. Hence, if a state of the state space is not reachable using transitions in the automaton, a trace $\pi \notin L(\mathcal{A})$ is found that can be output as a counterexample. From a user's perspective, the counterexample can now demonstrate a program trace $\pi = \langle s_0, \dots, s_{n-1}, s_n \rangle$, where the prefix $\langle s_0, \dots, s_{n-1} \rangle$ is recognized by the safety automaton, but the last transition $s_{n-1} \rightarrow s_n$ is not recognized. The corresponding states $q_0, \dots, q_{n-1}$ of the safety automaton for $s_0, \dots, s_{n-1}$ can then be inspected by the user.

6.3.3 *Extensions*

We introduce two extensions to safety automata for ease in writing the specification of certain properties. Firstly, we allow for *Alarm* and *Accept* states. That is, we allow states to be marked to immediately accept or reject a model. Accept states then allow to generate a *witness* on how to reach certain state, while alarm states allow to mark certain bad configurations. We do not allow the combination of alarm and accept states in one automaton. Otherwise, the order in which our model checking algorithm works would influence whether the alarm or accept is visited first. It could hence happen that the accepting state is visited first, verifying a model that would also reach an alarm state. Similarly, we do not allow for multiple accept states in one automaton, since the order in which they are visited would also depend on the actual model checking algorithm. We do, however, allow to check for the reachability of multiple alarm states.

Accept States and Alarm States

Priority of Transitions

Secondly, we also introduce optional *priorities* to the transitions so as to resolve ambiguity. Our definition of safety automata allows for overlapping conditions on guards, i. e., the conjunction of all conditions of the guards is satisfiable. In this case, more than one transition can be taken, which necessitates a non-deterministic choice by the model checker. If this behavior is not wanted, one would have to alter the guards accordingly, such that their intersection is empty. In the automaton in Fig. 16, e. g., multiple transitions have to check for $\neg$`error`, such that if an `error` signal is detected, the automaton will always assume the corresponding error state where the motor must be switched off. To resolve this ambiguity, PLCopen state

Algorithm 1 Model Checking Algorithm for Safety Automata

Input: Start state s_0
Input: Safety automaton $\mathcal{A} = (Q, q_0, \Sigma, \delta, \mathcal{I})$
Input: (optional) *AcceptState* or *AlarmStates*
Output: result / counterexample

if not $s_0 \models \mathcal{I}(q_0)$ **then**
  return "Counterexample", $\langle s_0 \rangle$
end if
worklist $\leftarrow$ **new** Queue
enqueue(*worklist*, $\langle s, \{q_0\} \rangle$)
markedTransitions $\leftarrow \{\}$
visited $\leftarrow \{\}$
while not empty(*worklist*) **do**
  $\langle s, Q' \rangle \leftarrow$ dequeue(*worklist*)
  $S' \leftarrow$ getSuccessors(s)
  for all $s' \in S'$ **do**
    $Q'' \leftarrow \{\}$
    for all $q' \in Q'$ **do**
      possibleTransitions $\leftarrow \{(q', G, q'') \in \delta \mid q'' \in Q, s' \models G, s' \models \mathcal{I}(q'')\}$
      for all $(q', G, q'') \in$ *possibleTransitions* **do**
        if $q'' \in$ *AcceptState* **then**
          return "Witness", $\langle s_0, \ldots, s' \rangle$ // cf. Sect. 6.3.3
        end if
        if $q'' \in$ *AlarmStates* **then**
          return "Counterexample", $\langle s_0, \ldots, s' \rangle$ // cf. Sect. 6.3.3
        end if
        $Q'' \leftarrow Q'' \cup q''$
        markedTransitions $\leftarrow$ *markedTransitions* $\cup \{(q', G, q'')\}$
      end for
    end for
    if $Q'' = \{\}$ **then**
      return "Counterexample", $\langle s_0, \ldots, s' \rangle$ // cf. Sect. 6.3.2
    end if
    if $\langle s', Q'' \rangle \notin$ *visited* **then**
      enqueue(*worklist*, $\langle s', Q'' \rangle$)
      visited $\leftarrow$ *visited* $\cup \langle s', Q'' \rangle$
    end if
  end for
end while
unmarkedTransisitons $\leftarrow \delta -$ *markedTransitions* // cf. Sect. 6.5.1
return "Valid", *unmarkedTransisitons*

diagrams introduce priorities for transitions, which determine in which order the transitions are checked. Only one transition can then be active at a time. By adopting such priorities, the example automaton would be simplified by checking the `error` with a higher priority than the other signals.

Implementation To implement priorities, we replace the inner loop in line 15 of Alg. 1: Instead of iterating over all possible transitions of one state of the safety automaton, we only consider the transitions with the highest priority: We set *possibleTransitions* to $\{(q', G, q'') \in \delta \mid q'' \in Q, s' \models G\}$ and select (q', G, q'') from *possibleTransitions* with highest priority. This is the only transition that will be taken into account. If $s' \not\models \mathcal{I}(q'')$ then a counterexample is returned (the highest priority transition violates the invariant). Observe that priorities and non-determinism are mutually exclusive for safety automata: Since the presence of priorities implies that only one transition of the safety automaton is selected in each step, non-determinism can no longer be expressed[1].

6.4 CHECKING PLCOPEN SAFETY FUNCTION BLOCKS

We implemented a model checker for safety automata and a graphical user interface to build safety automata into ARCADE.PLC. In an industrial cooperation, we then used safety automata to check the PLCOPEN safety function block (SFB) library implemented by ABB for the AC500 controller. We were able to verify 10 SFBs from this library using safety automata [23]. The results of this case study are presented in Tab. 5. The function block names are anonymized so as to protect implementation details by ABB. For these function blocks, the runtime of the model checking process was similar to the CTL model checking process: Most SFBs could be verified in seconds, verifying larger SFBs took minutes. The results further show that automata-based specification can be used without additional blow up of states. The crucial result is that we can now check the complete specification for the selected SFBs, since we were able to translate the complete PLCOPEN specification into a safety automaton.

Limitations For the other function blocks in the library, we were not (practically) able to write the specification as a safety automaton. The reason for this is that the PLCOPEN specification of some function blocks allows for macro states (states where a certain condition is indicated by a variable instead of a constant) or complex transition conditions that are explained in the accompanying documentation. Since safety automata only allow for simple guards in the transitions and simple invariants in the states, such complex specifications require a *flattening*, i. e., an enumeration into the possible values. For a high number of input variables this results in an explosion in the number of states in the safety automata. Here, it would be necessary to

1 Non-determinism only refers to the non-deterministic transitions in the safety automaton. The PLC model can of course still reflect non-deterministic behavior.

Function Block	#LOC	#States	#Transitions	Time	Memory
1	275	872	111 616	8.97 s	19.6 MB
2	312	665	85 120	8.56 s	19.0 MB
3	292	220	14 080	1.45 s	16.8 MB
4	307	2 074	530 944	50.83 s	28.9 MB
5	283	27	216	0.20 s	15.9 MB
6	229	134	4 288	0.33 s	16.2 MB
7	333	35	176	0.30 s	16.0 MB
8	210	134	4 288	0.27 s	16.2 MB
9	243	49	784	0.60 s	16.0 MB
10	238	49	784	0.70 s	16.0 MB

Table 5: Verification of an ABB PLCopen SFB library using Safety Automata

add hierarchical states as an extension to our user interface. The hierarchical states could then be flatten into normal states yielding a safety automaton.

Ambiguity of Macro States

We observed that the usage of macro states can also cause confusion or ambiguity. An example of such possibly confusing macro states is shown in Fig. 17, which depicts an excerpt from the `SF_TwoHandControlTypeIII` function block [94, p. 78]. Note that the states `C001`, `C002`, `C003` and `C004`, `C005`, `C006` have been merged into macro states (middle and upper right corner). Inputs `B1` and `B2` refers to the two buttons this function block is supposed to supervise. The exact meaning of `B1` and `B2`, however, has to be inferred from the accompanying documentation: For transitions, `B1` and `B2` indicate that the corresponding button is pressed. By way of contrast, in states `C001` to `C003` they refer to the configuration when the block was activated (i. e., when entering state `8001`). Finally, in states `C004` and `C005` their meaning change again and indicate that the respective button was *not* pressed before the time out.

While we manually flattened this automaton so as to write it as a safety automaton, we noticed further ambiguities apart from the inconsistent usage of `B1` and `B2` in the specification: In is unclear whether it is possible to switch in-between the states in a macro state, e. g., whether it is possible to switch to `C003` when entering through `C001` (i. e., if one button was pressed when the block was activated and then the second button is pressed). A similar issue can be observed with the other macro state. Here, the state `C004` should be asserted when button 1 is not pressed after the timeout in state `8006`. When button 1 is pressed in the same cycle where the timeout occurs, state `C006` should be asserted. It remains unclear whether state `C005` should be asserted if button 2 is released in the same cycle or if state `C005` should only reachable from state `8005`. While these ambiguities are not safety-critical, they could be avoided by the formal semantics of safety automata.

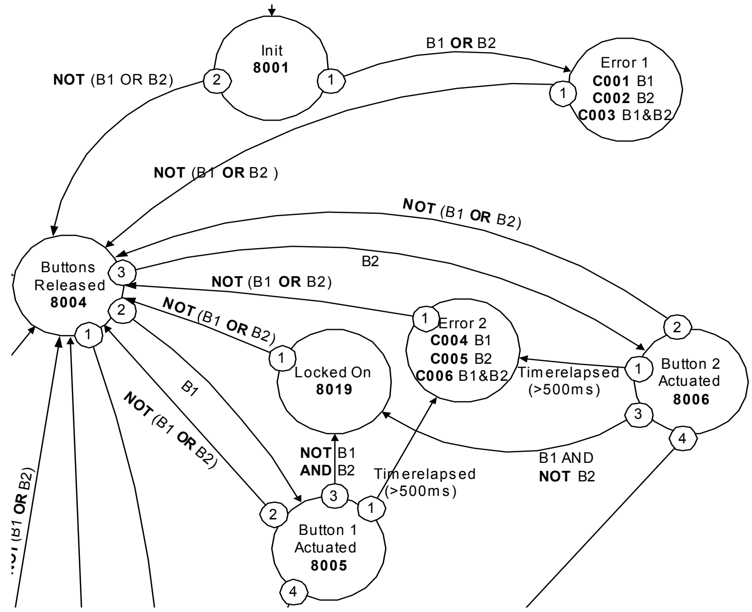


Figure 17: Excerpt from the `SF_TwoHandControlTypeIII` SFB taken from [94].

6.5 DETECTING OVER-SPECIFICATIONS

An *over-specification* is a (part of a) specification that is trivially fulfilled because its evaluation does not influence the final result. If, e. g., we require that B always happens after A, this requirement is fulfilled for a program that never exhibits A. In this case, we over-specified the program behavior and could prove the stronger requirement "A does never happen" without speaking about B. The key point here is that although the specification is fulfilled, it seems that either the program is not performing a crucial step (for which the specification is explicitly checking for) or that the author of the specification made a mistake.[2]

A very interesting application of safety automata arises from the fact that they offer a very natural way to detect such over-specifications.

6.5.1 *Detecting Over-Specifications in Safety Automata*

When we check an automaton-based specification, we can additionally check that each transition of the automaton has been taken at least once. This is performed by keeping track of which transitions have been taken while checking a safety automaton. We added this extension in line 22 in Alg. 1. Note that this extension will also detect whether there are unvisited states.

2 Of course, the specification could also intentionally be over-specified, so as to apply the same formula to a set of function blocks from a library that share similar behavior. This, however, is not a typical application.

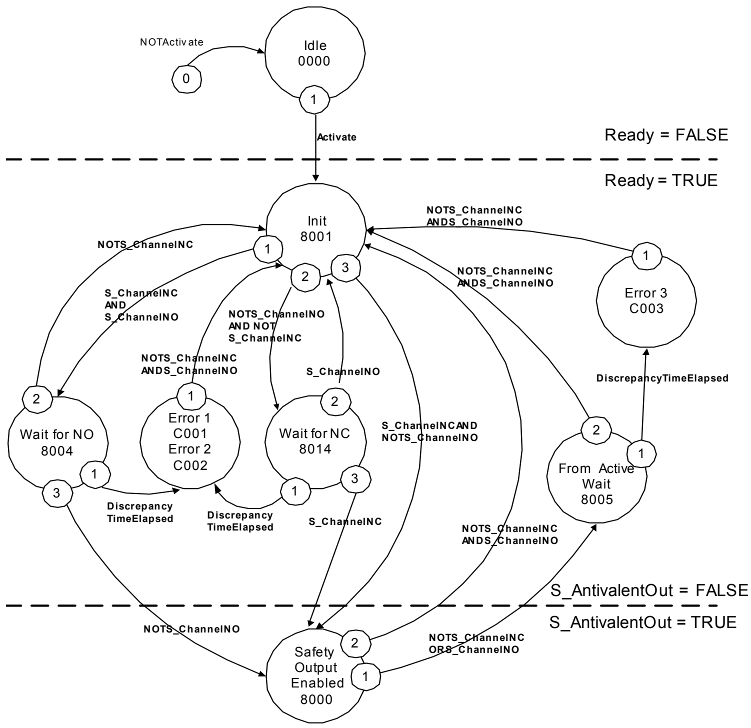


Figure 18: Erroneous specification for the SF_Antivalent SFB taken from [94].

The model checker for safety automata that we implemented in Arcade.PLC can hence report three different outcomes: (a) specification is valid, (b) specification is invalid, (c) specification is valid but over-specified. In case (b), a counterexample can be generated as seen in Sect. 6.3.2, similar to checking a CTL specification. In case (c), we can now present the unused transitions to the user. The user then has to decide whether these unused transitions are intentional, an error in the specification, or an error in the program.

6.5.2 *Detection of a Faulty Specification*

Over-Specification in SF_Antivalent

We enabled the technique to detect unused transitions in safety automata during model checking. We then rerun the case study from Sect. 6.4. Interestingly, it turned out that the safety automaton for the SF_Antivalent does include an over-specification. The automaton used by PLCopen is depicted in Fig. 18. This automaton can be directly interpreted as a safety automaton. After model checking this automaton against a manual implementation of the SF_Antivalent block, we got a warning that the transition from state 8000 (bottom) to state 8001 with priority 2 labeled NOT S_ChannelNC AND S_ChannelNO was never taken. The error here is that the priorities of both transitions are swapped. Hence, if S_ChannelNC is true and `S_ChannelNO` is false, the automaton assumes state 8005 and then state 8001, instead of going to state 8001 directly. This is not a safety problem per se but certainly an oversight.

Note that our `SF_Antivalent` implementation exactly follows the specification. Hence this error would not have been revealed using, e. g., CTL logic. Using the safety automaton, however, we can detect this error in the specification. Note further, that this is an error in the specification, and could thus be detected by analyzing solely this specification (without looking at the implementation). In a more general setting, however, our technique allows for finding over-specifications that are only revealed by looking at the program as well as the specification.

6.6 CONCLUDING DISCUSSION & FUTURE WORK

In this chapter we introduced safety automata and showed how PLC function blocks can be verified using this formalism. We introduced an algorithm for the verification that allows for extensions such as alarm and accept states. It is additionally able to detect over-specifications. The techniques presented can readily transferred to the verification of whole PLC programs.

6.6.1 *Automata Compared to* CTL

Safety automata were inspired by state diagrams used to specify PLCOPEN function blocks. They hence follow an established industry standard and are much easier to write and understand than comparable CTL formulae, although not all CTL properties can be expressed as safety automata. A key difference in specifying properties using safety automata compared to CTL is that safety automata can readily capture the full specification of a function block or a program. On the one hand, this simplifies the verification process, since only one (complete) specification has to be written. On the other hand, the core safety function (such as *an output must not be set if a certain input is set*) is no longer explicitly visible. In this case, we believe that it is advisable to formalize these safety function explicitly, either as automata or as CTL expressions.

Moreover, the possibility to detect over-specifications is a huge advantage when using safety automata. By enabling this warning, we were able to detect a wrong specification in a industry standard.

6.6.2 *Future Work*

A valuable extension to the safety automata described in this chapter is the possibility to express hierarchical states. In practice, such hierarchical states are necessary to model certain PLCOPEN state diagrams efficiently. To check hierarchical safety automata, either the model checker algorithm has to be extended or the hierarchies in these automata are flattened before applying the current algorithm.

Additionally, hierarchical states should have a possibility to formulate invariants in a variable way. To illustrate, the hierarchical stats shown in Fig. 17 on p. 82 (e. g., the state in the top right corner) have individual invariants on the variable `DiagCode` depending on which state was active before. To model all PLCopen function blocks in a succinct form, such as feature is necessary.

7

FAULT LOCALIZATION IN COUNTEREXAMPLES

When a model checker disproves an all-quantified formula, it provides a counterexample. Similarly, a witness can be provided for an existentially-quantified formula it can validate. In the context of PLCs, counterexamples provide the precise inputs necessary to reach a certain state of the program, usually after several cycles. An example of such a counterexample is given in Fig. 20 on p. 89. Often, a faulty state is non-trivial to reach, and thus counterexamples are regarded as "invaluable in debugging complex systems." [39]

Locating the Bug

Yet, even when a counterexample is given, the *actual* reason why a certain state is reachable might still be unclear. This is especially true when dealing with counterexamples instead of witnesses, i. e., the program *is* erroneous but it is unknown as to where exactly. Since the counterexample only refutes a given formula, it usually only captures the symptom of a bug, which can be hiding everywhere in the program. This situation is aggravated when the faulty code is executed many cycles before the erroneous behavior is detected. This can happen, e. g., when local variables are assigned wrong values: If these variables influence the visible program behavior only in later PLC cycles, the erroneous assignment is hard to track down.

7.1 APPROACH

In this chapter, we will explore several techniques that try to automatically highlight possibly causes of a counterexample. These techniques are, in their core, heuristic approaches that might not always succeed. Especially when a bug is caused by missing or unimplemented code, it is hard to locate the exact point where this has happened. As we will show in this chapter, however, in many typical cases the techniques can pinpoint the problem down to a small number of possible error candidates, sometimes even to the exact error location. Additionally, these techniques can provide possible fixes that make the program correct.

Trace Comparison

The key idea of these approaches is that each line of the program can be executed in different contexts, i. e., each line can be part of different program execution traces. Some of these traces end in an error state, i. e., they are a counterexample for a specification. Other traces, however, do not exhibit erroneous behavior. One way to select possible error candidates is to compare good traces against bad traces.

```
FUNCTION_BLOCK Antivalent
VAR_INPUT Activate, NC, NO: BOOL; END_VAR
VAR_OUTPUT Ready, Out: BOOL; END_VAR
VAR DiagCode: WORD; END_VAR

IF NOT Activate THEN
    DiagCode := 16#0000;
ELSE
    CASE DiagCode OF
    16#0000:
        IF Activate THEN
            DiagCode := 16#8001;
        END_IF;
    16#8001:
        IF NC AND NO THEN
            DiagCode := 16#8004;
        ELSIF NOT NC AND NOT NO THEN
            DiagCode := 16#8014;
        ELSIF NC AND NOT NO THEN
            DiagCode := 16#8000;
        END_IF;
    16#8004:
        IF NOT NC THEN
            DiagCode := 16#8001;
        ELSIF NOT NO THEN
            DiagCode := 16#8000;
        ELSE
            DiagCode := 16#C001;
        END_IF;
    16#C001,
    16#C002:
        IF NOT NC AND NO THEN
            DiagCode := 16#8000;
        END_IF;
   16#8014:
        IF NO THEN
            DiagCode := 16#8001;
        ELSIF NC THEN
            DiagCode := 16#8000;
        ELSE
            DiagCode := 16#C002;
        END_IF;
    16#C003:
        IF NOT NC AND NO THEN
            DiagCode := 16#8001;
        END_IF;
    16#8005:
        IF NOT NC AND NO THEN
            DiagCode := 16#8001;
        ELSE
            DiagCode := 16#C003;
        END_IF;
    16#8000:
        IF NOT NC AND NO THEN
            DiagCode := 16#8001;
        ELSIF NOT NC OR NO THEN
            DiagCode := 16#8005;
        END_IF;
    END_CASE;
END_IF;
CASE DiagCode OF
16#0000:
    Ready := FALSE; Out:= FALSE;
16#8000:
    Ready := TRUE; Out := TRUE;
ELSE:
    Ready := TRUE; Out:= FALSE;
END_CASE;
```

Figure 19: A faulty `Antivalent` implementation

We present this technique in Sect. 7.3. Alternatively, we can automatically check whether the execution of a certain program line is a necessary or sufficient condition to violate the specification. We perform this step by augmenting the specification with propositions about the executed lines. This technique is presented in Sect. 7.4.

Candidate Exclusion

7.2 MOTIVATING EXAMPLE

We motivate our approach with the example program shown in Fig. 19. This program implements an `Antivalent` block (cp. Sect. 6.5.2 and Sect. 8.6.3), i. e., a monitor that the safety signal `NC` (normally closed) is true and `NO` (normally open)

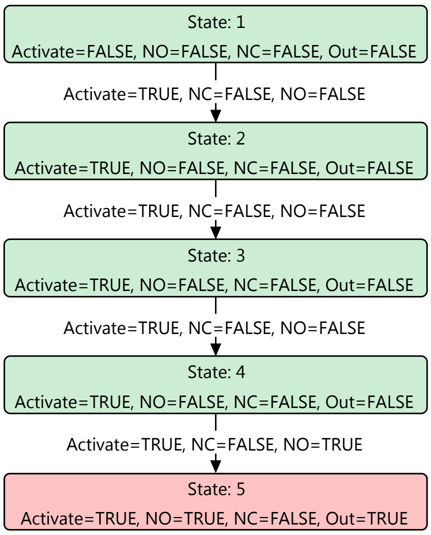


Figure 20: Counterexample for the program in Fig. 19 refuting formula (2) as generated by ARCADE.PLC

is false, and sets `Out` only if this is the case. Typically, such blocks also have a timer to monitor a discrepancy time, where the signals of the safety inputs are not yet stable. For exposition, we omitted this functionality here. Similarly, we also stripped an error output, which would signal that the inputs are not consistent.

The key safety property of this block written in CTL is:

$$AG\ \varphi \tag{1}$$

where

$$\varphi \quad := \quad (\neg\texttt{Activate} \vee \texttt{NO} \vee \neg\texttt{NC}) \implies \neg\texttt{Out} \tag{2}$$

We can check the property (1) using the techniques detailed in Chap. 4. Since there is a slight mistake in the state machine implementation in Fig. 19, this specification is violated and the model checker generates a counterexample. A counterexample for the specification that was generated using ARCADE.PLC is depicted in Fig. 20. Observe that the counterexample is non-trivial: Although this is the shortest trace that violates the specification[1], four invocations of the function block were necessary. The programmer now would have to inspect the counterexample trace carefully to locate the actual error. Note that although the counterexample provides all the necessary information, it is still not obvious where the error is in the program. This situation gets worse when we consider a real (larger) `Antivalent` block instead of the stripped-down example shown here.

1 Proof omitted, but easy to see once we reveal the cause of the bug.

To find the error location, one might be tempted to evaluate (2) for each program location instead of the observable behavior of the function block at the end of the cycle. This, however, would only indicate that the violation occurs in line 65, which obviously is not the erroneous program line. To manually find the error, one now has to backtrack how line 65 can be reached (leaving error candidates in lines 20, 26, 33, 39 and 45). Careful inspection of the counterexample would then reveal that line 33 is actually the culprit: `DiagCode` should have be set to `16#8001` there. In the next section, we describe how this fault location can be detected automatically.

7.3 TRACE COMPARISON

The key idea of *trace comparison* technique described in this section is to sample good (i. e., non-violating) runs and then compare them syntactically against a counterexample. As we will show, this technique is often very effective and can sometimes even provide corrections for a program. In Sect. 7.3.2, we will start by first looking at the last cycle only. Then, we will extend the technique to full traces so as to detect faulty code executed in earlier cycles in Sect. 7.3.3.

7.3.1 *Preliminaries*

We first recall the definition of a trace:

Trace

Definition 7.1: A *trace* $\pi = \langle s_0, \dots, s_n \rangle$ is a sequence of program states s_i. Each state represents the PLC at the end of one cycle (cp. Sect. 2.3). We write $|\pi| = n$ for the length of the trace.

Each transition between a state s_i and s_{i+1} represents thus one cycle, i. e., one invocation of the program. While executing the program, intermediate states are generated. Since I/O is only performed at the beginning/end of the cycle, these intermediate states are non-observable. Hence, the intermediate states are not part of the state space. They are, however, useful in detecting an error location, since an error occurs at one of the intermediate steps of the program.

Intermediate Trace

Definition 7.2: Let $\pi = \langle s_0, \dots, s_n \rangle$ be a trace. An *intermediate trace* between state s_i and s_{i+1} is a sequence $\tau_{s_i \to s_{i+1}} = \langle \iota_0, \dots, \iota_m \rangle$ of intermediate instructions ι_j that were executed in that cycle. Each intermediate instruction ι_j can represent, e. g., an instruction of our IR, a line of ST code, an IL instruction, etc. For $i < j$ we generalize this notation and write $\tau_{s_i \to s_j}$ for the concatenation of the intermediate traces $\tau_{s_i \to s_{i+1}}, \tau_{s_{i+1} \to s_{i+2}}, \dots, \tau_{s_{j-1} \to s_j}$. Further, let $\tau_\pi := \tau_{s_0 \to s_n}$.

7.3.2 *Analysis of the Last Cycle*

Let $\pi_c = \langle s_0, \ldots, s_{n-1}, s_n \rangle$ be a counterexample trace for a safety property φ, i. e., $\varphi \models s_i$ for $1 \leq i \leq n-1$ but $\varphi \not\models s_n$. An example for such a trace is shown in Fig. 20: States 1–4 fulfill (2) whereas state 5 violates the formula.

In this section, we first consider the case that the error occurs in the same cycle where it is detected, i. e., in the transition from state 4 to state 5. We are now interested in *similar* traces that do fulfill the formula. Since we assume in this section that the fault occurred in the last transition, we look at neighbors that only deviate in the last step:

Direct Neighbor Trace

Definition 7.3: Let $\pi = \langle s_0, \ldots, s_{n-1}, s_n \rangle$ be a trace. A *direct neighbor trace* of π is a trace $\pi' = \langle s'_0, \ldots, s'_{n-1}, s'_n \rangle$ with $s_i = s'_i$ for $0 \leq i \leq n-1$, i. e., π and π' deviate at most in the last state.

We define the set $G = \{\pi \mid \pi \text{ is direct neighbor trace of } \pi_c \text{ and } \pi \models \varphi\}$ as the *good* neighbors of a counterexample π_c. We can easily construct G by inspecting the direct neighbor traces in the state space or by regenerating the successors at the second to last state (cp. Sect. 2.5.1). To select the closest neighbor π_c from the good neighbors in G, we need to introduce some kind of metric between traces. This metric will be defined over the intermediate traces. Here, we choose the Levenshtein distance [81], which is a metric for measuring the difference between two sequences:

Levenshtein Distance

Definition 7.4: Let $S_1 \in \Sigma^n$ and $S_2 \in \Sigma^m$ be sequences over a common alphabet Σ, with n, m their respective lengths and $S_x[y]$ being the y^{th} character of S_x. The *Levenshtein distance* $\text{lev}(S_1, S_2)$ between S_1 and S_2 is then inductively defined as $\text{lev}(S_1, S_2, n, m)$ with:

$$\text{lev}(S_1, S_2, i, j) := \begin{cases} \max\{i, j\} & \text{if } \min\{i, j\} = 0 \\ \min \left\{ \begin{array}{l} \text{ins}(S_1, S_2, i, j) \\ \text{del}(S_1, S_2, i, j) \\ \text{subst}(S_1, S_2, i, j) \end{array} \right\} & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \text{ins}(S_1, S_2, i, j) &:= \text{lev}(S_1, S_2, i, j-1) + 1 \\ \text{del}(S_1, S_2, i, j) &:= \text{lev}(S_1, S_2, i-1, j) + 1 \\ \text{subst}(S_1, S_2, i, j) &:= \begin{cases} \text{lev}(S_1, S_2, i-1, j-1) & \text{if } S_1[i] = S_2[j] \\ \text{lev}(S_1, S_2, i-1, j-1) + 1 & \text{if } S_1[i] \neq S_2[j] \end{cases} \end{aligned}$$

The Levenshtein distance can be interpreted as the minimum number of character insertions, deletions or substitutions that are necessary to transform one sequence into the other. It can efficiently be calculated using a dynamic programming algorithm whose runtime is in $\mathcal{O}(mn)$ as shown by [120]. Additionally to the Levenshtein distance, this algorithm also returns the concrete operations (i. e., substitutions, insertions, deletions) necessary to transform the sequences.

Distance between Intermediate Traces The distance between intermediate traces τ_0, τ_1 can now be interpreted as the Levenshtein distance of the syntactic comparison of the intermediate instructions:

$$\mathrm{dist}(\tau_0, \tau_1) := \mathrm{lev}(\tau_0, \tau_1) \tag{3}$$

This approach is similar to [76]. Note that distance does not take any semantic similarities of instructions or the context into account, but works purely on the syntactic comparison. Hence, our distance is consistent under certain abstractions that do not merge different program paths. This dovetails nicely with the abstraction refinement we describe in Sect. 4.4.1, which motivates the following:

Abstract Traces *Proposition 7.1:* It is sufficient to look at the abstracted 1-neighbor traces according to the abstractions described in Sect. 4.4.1.

Proof. We compare the instructions of an intermediate trace. Each path through the program will lie in the same equivalence class according to this metric, independent of the values of the variables. The techniques described in Sect. 4.4.1 will refine values of variables until they describe a single path through the program. Hence, each abstract trace still represents a single path through the program. □

Back to the Example We can now apply this approach to the example program. Since the program has three inputs, the counterexample trace has $2^3 = 8$ direct neighbors. Looking at state 4, we see that there are three equivalence classes of intermediate traces (by syntactically comparing the executed lines):

1. `Activate=TRUE` and `NC=FALSE` and `NO=TRUE`: This corresponds to the single counterexample trace shown in Fig. 20.

2. `Activate=FALSE`: In this case, the values of `NC` and `NO` do not matter, so we have 4 traces.

3. `Activate=TRUE` and (`NC=TRUE` or `NO=FALSE`): 3 traces.

All good neighbors of the counterexample are in classes 2 and 3, since the specification is valid there. It is easy to see that the distance between class 1 and class 3 is 1: The only difference is line 33. The distance between class 1 and class 2 is certainly higher. Hence, we conclude that problem might lie in the difference between class 1 and 3, which is line 33. It is indeed the case that this line is the faulty one.

The Levenshtein distance also allows us to extract a possible fix for this problem, which is "delete line 33". Interestingly, this yields a program that satisfies (2) and does not just block this single counterexample. Yet, this modified program certainly does not reflect the programmers *intent* here and would fail to satisfy other specifications. To summarize, we learn two lessons from this example. Firstly, the purely syntactic suggestions for possible corrections drawn from the Levenshtein distance are sometimes good enough to fulfill a specification. Secondly, although a program fulfills a specification, it can be far away from the desired program. The

A Faulty Correction

crucial part here is that the modified program is, from the perspective of the specification, as good as it can get, so we will likely have to rely on heuristic techniques here that require user feedback.

We will now first generalize the approach and then present a case study, which will helps us to assess accuracy of this technique.

7.3.3 *Analysis of a Trace*

Sometimes, looking at the direct neighbors is not sufficient. It could be, e. g., that the good neighborhood G is empty because all direct neighbors violate the specification. In general, the effect of faulty instruction could only be observable several cycles later. For such cases, we thus want compare intermediate traces from earlier cycles. We thus have to generalize the distance: For traces π_0 and π_1 (w. l. o. g.: $|\pi_0| \leq |\pi_1|$) we define the distance as the sum over the pair-wise intermediate traces:

Distance between Traces

$$\mathrm{dist}(\pi_0, \pi_1) := \sum_{i=0}^{|\pi_0|-1} \mathrm{dist}(\tau_0^i, \tau_1^i) \; + \sum_{i=|\pi_0|}^{|\pi_1|-1} |\tau_1|, \tag{4}$$

where τ_j^i is the i^{th} intermediate trace of π_j.

Alternatively, one could define this distance as the distance between the complete intermediate traces τ_{π_0} and τ_{π_1}. We use the cycle-wise definition here because it will simplify considerations about possible corrections of a fault in Sect. 7.3.4. For long counterexamples this is also faster because computing the Levenshtein distance scales quadratic in the number of states. Since the Levenshtein distance is a metric where the triangle inequality holds, our definition yields slightly higher distances than comparing the complete intermediate trace.

We are now looking for the nearest satisfying neighbor trace of π_c, i. e., a trace π with $\pi \models \varphi$ and $\mathrm{dist}(\pi, \pi_c)$ minimal. To iteratively search for such a neighbor, we first introduce the *n-distance*: Let $\pi_1 = \langle s_0, \ldots, s_{m_1} \rangle$, $\pi_2 = \langle s'_0, \ldots, s'_{m_2} \rangle$ be traces and $n \leq \min(m_1, m_2)$. The *n-distance* is the distance of the prefix of length n of π_1 and π_1, i. e.:

n-distance

$$n\text{-dist}(\pi_1, \pi_2) := \mathrm{dist}(\langle s_0, \ldots, s_n \rangle, \langle s'_0, \ldots, s'_n \rangle) \tag{5}$$

Using this function, algorithm 2 searches for the closest neighbor, starting at s_0. This algorithm makes use of a priority queue Q, which stores all traces according to their n-distance to π_c. In each iteration, all new traces obtained by extending the last state of the closest neighbor are added to the queue. If a trace with length $|\pi_c|$ is found that fulfills φ, it is by definition the closest non-violating trace.

7.3.4 *Correction Candidates*

A side-effect of using the Levenshtein distance is that we also get possible corrections for faulty traces. A correct can either be an insertion, a replacement or

Algorithm 2 Find nearest non-violating trace [21]

Input: Formula φ
Input: Counterexample $\pi_c = \langle s_0, \dots, s_n \rangle$ for φ
Output: Nearest non-violating trace π

$Q \leftarrow$ **new** PriorityQueque
assert $s_0 \models \varphi$
enqueue$(Q, (\langle s_0 \rangle, 0))$
while not isEmpty(Q) **do**
  $(curTrace, minDist) \leftarrow$ dequeue(Q)
  if $|curTrace| = |\pi_c|$ **then**
    // same length as π_c and satisfies φ
    return *curTrace*
  end if
  curState $\leftarrow$ last state of *curTrace*
  successors $\leftarrow$ createCycleSuccessors$(curState)$
  for all *newState* in *successors* **do**
    newTrace $\leftarrow$ concat$(curTrace, newState)$
    if *newTrace* $\models \varphi$ **then**
      // only follow traces that satisfy φ
      $n \leftarrow |newTrace|$
      $dist \leftarrow n$-dist$(\pi_c, newTrace)$
      enqueue$(Q, (newTrace, dist))$
    end if
  end for
end while
return "no trace found"

a deletion of an instruction. These corrections are based purely on syntactic differences, i. e., they do not take into account whether a correction makes sense semantically or even if it is a legal construct. We can, however, rerun the model checking process and thus automatically check whether a correction actually fixes the problem (or at least yields a legal program that satisfies the specification).

If the model checking run then succeeds, we can be reasonably sure to have found the actual problem in the program. This correction can, however, still be unsuitable for allowing the program to fit the programmers intention. As we have seen in the example program, e. g., the correction can be a deletion of a wrong statement, which then allows the program to fulfill the specification, but still with a lack of the required functionality.

Program	Sev.	Change	Loc	$\lvert\pi_c\rvert$	$\lvert C\rvert$	#Hit	Time
`EnableSwitch`	Simple	Ass.	120	15	5	1	< 0.1 s
`EnableSwitch`	Medium	Ass.	120	80	4	1	0.5 s
`EnableSwitch`	Simple	Add.	121	81	4	0	< 0.1 s
`EmergencyStop`	Medium	Ass.	115	67	3	1	0.1 s
`EmergencyStop`	Simple	Branch.	115	13	6	1	< 0.1 s
`EmergencyStop`	Complex	Miss.	111	16	9	0	< 0.1 s
`SafetyRequest`	Complex	Miss. & Ass.	140	193	6	2	0.5 s
`ModeSelector`	Simple	Ass.	155	26	5	1	0.1 s
`GuardMonitoring`	Complex	Add.	110	17	3	2	< 0.1 s

Table 6: Case Study of the Trace Comparison technique [21]

7.3.5 *Case Study*

To judge the effectiveness of the trace comparison technique, we perform a case study based on the PLCopen safety function blocks [94], where we manually induced some errors. For each block, we selected some CTL property. Then, we altered the code of each block to force a violation of these properties. We considered the errors (1) assignment of a wrong value, (2) wrong branch condition, (3) missing code, and (4) extra (erroneous) code. Bugs can also comprise multiple lines with multiple bug types.

We classify the bugs according to their severeness: *Simple* bugs incorporate a change of a single line or assignment, e. g., the assignment $var :=$ `TRUE` becomes $var :=$ `FALSE`. Further, we consider *simple* bugs to have a direct impact on the violation of the property, i. e., the execution of the faulty line will cause the violation in the same cycle. *Medium* bugs may involve one to several lines of code and indirectly influence the violation of the property. Bugs involving multiple lines, strongly altering the program behavior, are defined as *complex* bugs. The case studies were performed on a MacBook Pro (Mid 2010) with an Intel Core i5 (2.53 GHz) and 8 GiB RAM. The results of the trace comparison approach are shown in Tab. 6. This table includes:

- Severeness of the bug (Sev.)
- Involved bug types (Change)
- Number of program lines (Loc)
- Length of extended counterexample ($|\pi_c|$)
- Number of error proposals ($|C|$)
- Number of correctly found errors (#Hit)

- Runtime (Time)

Altered or extra lines are counted as *Hit*, if they are marked as an error. For missing code and wrong branch conditions, we define a *range* to decide whether a missing line is correctly identified as an error. If the missing line is marked with an offset of at most 2 instructions in the execution order, we consider the bug to be found. For wrong branch conditions, typically only one branch initiates the error. Thus, we consider the bug to be found if every instruction in the erroneous branch is marked. This trade-off is necessary, since there may exist program traces that execute the branch condition without violating the formula.

7.3.6 *Discussion*

The proposed error candidates significantly reduce the effort to locate the bugs in the program. Even in case of an inaccurate candidate, it still provides a good starting point for locating the bug manually. Only in two instances the proposed error candidates were not useful. In one case, this was caused by missing code, which, in principle, is harder to detect than wrong code. Offering results in far less than a second in most cases, our technique requires low effort and is directly applicable as a starting point for debugging. The results of checking the `SafetyRequest` function block outline the efficient simplification of the search space. In this case, the extended counterexample yields 193 locations, which makes manual examination infeasible. Our technique reduces the vast search space to six lines containing two errors.

The quality of error correction proposals, however, depends highly on the non-violating trace that is used to compare to the counterexample trace. An ideal non-violating trace, e. g., would differ in its execution only by the erroneous instructions. Many specifications we checked in our case study were of the form $\varphi \Rightarrow \psi$. A counterexample will thus fulfill φ while violating ψ. A non-violating trace can, however, fulfill the formula by violating φ. Though syntactically similar, the resulting trace might greatly differ in its semantics. Corrections proposed by the *Levenshtein distance* might thus be misleading. In literature, this is referred to as the *multiple nearest witnesses problem* [76]. We alleviate this problem by offering the programmer the possibility to manually skip traces that do not yield sufficient explanation. Alternative non-violating traces produced this way may offer more error candidates, while giving a better explanation of the error.

As we have seen in the case study, the trace comparison technique works fast if we can enumerate the paths using the techniques described in Chap. 4. If, however, the programs are more complex and it is no longer possible to efficiently inspect all paths, a different approach is needed. In the next section we will describe another approach that is purely based on repeated model checker calls, and hence, can readily be applied to other abstraction techniques such as the predicate abstraction described in Chap. 5.

7.4 CANDIDATE EXCLUSION

In this section, we describe a different heuristic to locate faulty statements in counterexamples. The key idea of this technique is to test whether the execution of certain lines is a *necessary* or a *sufficient condition* for the violation to occur. We can do this by augmenting the original specification, adding a clause that tests whether a certain line number was executed. We therefore introduce the atomic proposition $\mathcal{L}_{\text{cycle}}(\ell)$ to denote that line ℓ was executed in the last cycle to our specification mechanism[2]. The model checker can then be used to test, e. g., whether certain lines are always part of a counterexample. As in the last section, we assume that φ is an invariant and π_c a counterexample for φ with intermediate trace τ_{π_c}.

Sufficient Condition

If we want to test whether the execution of a line ℓ is a sufficient condition to induce a violation of φ, we can check the following formula:

$$\text{Reach}_{\text{suf}}(\varphi, \ell) := AG\ (\mathcal{L}_{\text{cycle}}(\ell) \implies \neg\varphi). \tag{6}$$

A counterexample of $\text{Reach}_{\text{suf}}(\varphi, \ell)$ indicates that the line ℓ can be executed without violating φ in the same cycle, i. e., ℓ is not sufficient. We can also ask whether the execution of ℓ is necessary for the violation:

Necessary Condition

$$\text{Reach}_{\text{nec}}(\varphi, \ell) := AG\ (\neg\varphi \implies \mathcal{L}_{\text{cycle}}(\ell)). \tag{7}$$

If this formula is violated, we obtain a counterexample that hits other lines than ℓ and eventually also violates φ. Formulae (6) and (7) can now be used to iteratively test each line of a counterexample:

Sufficient and Necessary Error Candidates

Definition 7.5: Let L be the set of line numbers of the program. The *necessary error candidates* set $\text{Cand}_{\text{nec}}(\varphi)$ and *sufficient error candidates* set $\text{Cand}_{\text{suf}}(\varphi)$ for a property φ are defined as:

$$\text{Cand}_{\text{nec}}(\varphi) := \{\ell \in L \mid \text{Reach}_{\text{nec}}(\varphi, \ell) \text{ is true}\} \tag{8}$$

$$\text{Cand}_{\text{suf}}(\varphi) := \{\ell \in L \mid \text{Reach}_{\text{suf}}(\varphi, \ell) \text{ is true}\} \tag{9}$$

Worked Example

We can now apply this technique to our worked example, with φ defined as in (2). We obtain that $\text{Cand}_{\text{suf}}(\varphi) = \{33\}$ (since all other lines can also be executed in good contexts). This is exactly the faulty line and hence a more precise result compared to the trace comparison technique. For the necessary error candidates, we obtain $\text{Cand}_{\text{nec}}(\varphi) = \{6, 9, 32, 33, 61, 65\}$. We can thus conclude that the execution of line 33 is necessary and sufficient to cause the error, which is a strong result.

We will now extend this conceptually simple technique in different directions to be more powerful in practice.

2 The labeling of the states with such atomic propositions is performed without overhead while building the state space.

7.4.1 *Testing Multiple Lines at Once*

First, we want to speed up the process of calculating $\text{Reach}_{\text{suf}}(\varphi)$ by testing multiple lines at once. If L_0 is a set of lines, we can check whether each of the lines in L_0 is sufficient to induce a counterexample by checking $\text{Reach}_{\text{suf}}(\varphi, L_0)$, where:

$$\text{Reach}_{\text{suf}}(\varphi, L) := AG\ (\bigvee_{\ell \in L} \mathcal{L}_{\text{cycle}}(\ell) \implies \neg\varphi) \tag{10}$$

If this formula is violated, we obtain counterexample π. Assume that L_π is the set of lines covered in the last cycle of π. We now know that all lines in L_π are not sufficient to induce an error. We set $L_1 := L_0 - L_\pi$ and repeat this process, checking $\text{Reach}_{\text{suf}}(\varphi, L_1)$. This process will eventually converge to a set L': In each step we remove a line, otherwise the formula $\text{Reach}_{\text{suf}}(\varphi, L)$ becomes true. Note that the formula is trivially true for an empty set. Similarly, we can test whether multiple lines are necessary using

$$\text{Reach}_{\text{nec}}(\varphi, L) := AG\ (\neg\varphi \implies \bigwedge_{\ell \in L} \mathcal{L}_{\text{cycle}}(\ell)) \tag{11}$$

and an analogue process.

7.4.2 *Testing Multiple Cycles*

Thus far, we used the proposition $\ell \in \mathcal{L}_{\text{cycle}}$ to test whether a line was executed in the last cycle before the formula was violated. Similar to the trace comparison, a canonical question now is relating to how we can extend this technique to lines that were executed in the past.

We can generalize the necessary condition and test for lines that are necessary to execute to induce an error. To do so, we check the opposite condition: Is there a path that violates φ without executing a line ℓ:

$$\text{Reach}^*_{\text{nec}}(\varphi, \ell) := \neg E\ (\neg\mathcal{L}_{\text{cycle}}(\ell)\ U\ \neg\varphi). \tag{12}$$

We define $\text{Cand}^*_{\text{nec}}(\varphi)$ analogue to Def. 7.5 using this advanced criterion.

Similarly, we can try to generalize $\text{Reach}_{\text{suf}}(\varphi, \ell)$ to check whether the execution of ℓ will always end eventually in an error state:

$$\text{Reach}^*_{\text{suf}}(\varphi, \ell) := AG\ (\ell \in \mathcal{L}_{\text{cycle}} \implies AF\ \neg\varphi). \tag{13}$$

This formula, however, has not been proven useful in practice: Since a counterexample usually depends on (non-deterministic) inputs, not all continuations after a faulty line was executed will eventually end in an error state: Some successor states will, e. g., just loop, waiting for new inputs. This is sufficient to invalidate (13). Weakening the second part of this formula to $EF\ \neg\varphi$ also does not help: Due to their reactive nature, typical PLC programs and function blocks are *resettable*.

Resettable Programs

```
IF cond1 THEN
        (* Block 1 *)
        ...
ELSE
        (* Block 2 *)
        ...
END_IF;
IF cond2 THEN
        (* Block 3 *)
        ...
ELSE
        (* Block 4 *)
        ...
END_IF;
```

Figure 21: Example

That means that it is always possible to reset the program to the initial state, by, e. g., setting some `Reset`, `Activate` or `EN` input. It is hence always possible to (a) execute each line of the program, then (b) reset the program (c) reach the error state. Thus, each line of the program would be a sufficient error candidate under this condition. We will, therefore, conclude that all sufficient error candidates occur in the last cycle before the violation and so only test necessary lines using $\mathrm{Reach}^*_{\mathrm{nec}}(\varphi, \ell)$.

7.4.3 *Coincidental Correctness & Preconditions*

Sometimes, a bug is not caused by a single faulty statement but by multiple (wrongly interacting) statements. In this case, locating the exact necessary and sufficient error lines can be more convoluted. To exemplify, Fig. 21 shows a program excerpt with four basic blocks. Each cycle, condition `cond1` switches between block 1 and block 2 and condition `cond2` switches between block 3 and block 4. We assume that a bug only manifests itself if an instruction from block 1 and an instruction from block 3 is executed. In this case, neither instructing is sufficient to trigger the bug, because we can execute block 1 and then block 4, or block 2 and then block 3. Such behavior is called *coincidental correctness* [6]. What happens in this case is that $\mathrm{Cand}_{\mathrm{suf}}(\varphi)$ is an empty set.

Coincidental Correctness

A way to handle such cases is to allow *conjunctions* of executed lines as sufficient error candidates. Note, however, that trying all possible conjunctions of lines is computationally expensive, even when restricting this to the candidates already found necessary.

Therefore, we use a different approach: We alter the definition of sufficient error candidates to take into account *preconditions*. The key idea here combines two insights: (a) The last intermediate trace of a counterexample contains the sufficient

Preconditions

error candidates and (b) which instructions are executed in the last trace is—due to the cyclic scanning mode—completely determined by a precondition, i. e., the values of input variables and the local variables of the program at the beginning of the cycle.

To exemplify, consider the last transition of the counterexample shown in Fig. 20. Its precondition is composed of two factors: Firstly, the program has to be in state $\psi^c_M := \texttt{Diagcode} = \texttt{16\#C002}$ (this corresponds to the penultimate state of the counterexample). Then, the precondition $\psi^c_I := \texttt{Activate} \wedge \neg\texttt{NC} \wedge \texttt{NO}$ on the inputs entails that the last (violating) state is reached. Observe that $\psi^c := \psi^c_I \wedge \psi^c_I$ entails the violation: ψ^c implies that all lines are error candidates, since their execution will inevitable yield to an error. The key idea now is that we can weaken the precondition to exclude as many lines as possible as potential error candidates. If the precondition is as weak as possible, but there are still error candidates left, it is likely that the error is caused by these candidates.

Sufficient Error Candidate under Precondition

In the following, we always decompose a precondition $\psi := \psi_I \wedge \psi_M$ into a precondition ψ_I on the input configuration and ψ_M on the local variables (cp. Def. 2.2 in Sect. 2.5.1). Given precondition $\psi := \psi_I \wedge \psi_M$, we can test whether a line is a *sufficient error candidate under* ψ using:

$$\mathrm{Reach}^{\psi_I,\psi_M}_{\mathrm{suf}}(\varphi,\ell) := AG\ (\psi_M \implies AX\ (\psi_I \wedge \mathcal{L}_{\mathrm{cycle}}(\ell) \implies \neg\varphi)). \tag{14}$$

What this formula expresses is that for all successor states of ψ_M being true, where ψ_I is true and ℓ was executed, φ is violated. We define $\mathrm{Cand}^{\psi_I,\psi_M}_{\mathrm{suf}}(\varphi)$ analogue to Def. 7.5. This set can be efficiently determined using an algorithm similar to the one described in Sect. 7.4.1. Observe that $\mathrm{Cand}^{true}_{\mathrm{suf}}(\varphi) = \mathrm{Cand}_{\mathrm{suf}}(\varphi)$.

Computing Preconditions

We are now interested in sufficiently weak preconditions such that $\mathrm{Cand}^{\psi_I,\psi_M}_{\mathrm{suf}}(\varphi)$ is not empty. To compute such preconditions, we use the following approach:

1. Let ψ be the precondition of the last step of a counterexample written in *conjunctive normal form* $\psi = \bigwedge_i \psi_i$. Observe that $\mathrm{Cand}^{\psi}_{\mathrm{suf}}(\varphi)$ is not empty, since ψ fixes a path of the program.

2. Set $\psi^0 := true$. For each i, repeat:

3. Set $\psi^- := \psi^i \wedge \bigwedge_{j>i} \psi_j$. If $\mathrm{Cand}^{\psi^-}_{\mathrm{suf}}(\varphi) = \{\}$ then set $\psi^{i+1} := \psi^i \wedge \psi_i$ otherwise set $\psi^{i+1} := \psi^i$.

This algorithm iteratively weakens the precondition ψ by removing conjuncts. A conjunct is not removed if its removal would cause the set of candidates to become empty. The final ψ^n (for n conjuncts) is hence a weaker precondition such that $\mathrm{Cand}^{\psi^n}_{\mathrm{suf}}(\varphi)$ is not empty. It is not necessarily the weakest precondition, however we only have to try each conjunct once. We favor this faster approach against the weakest precondition because of the heuristic nature of potential error candidates.

Another benefit of this extension is that we can now also handle violations of specifications that depend not only on the lines executed but also on certain values of variables: If, e. g., a specification is only violated if a certain variable v is greater than zero (but the control flow of the program does not depend on this property), then $\text{Cand}_{\text{suf}}(\varphi)$ is empty. The error candidates under precondition $\text{Cand}_{\text{suf}}^{v>0}(\varphi)$, however, can now reveal the error.

7.4.4 *Multiple Necessary Error Candidates*

Multiple Bugs

It can also happen that we have two bugs in the program, e. g., in block 1 and block 2 of Fig. 21. If these two bugs happen to cause the same violation, then unfortunately neither the execution of block 1 nor block 2 is a necessary condition for the bug. Hence, $\text{Cand}_{\text{nec}}(\varphi)$ does not contain the necessary lines that cause either violation. In this case, the technique fails to extract either of the bugs from a counterexample.

7.4.5 *Case Study*

We repeated the case study from Sect. 7.3.5 using the candidate exclusion technique. The results are presented in Tab. 7. All programs are evaluated in the same way as we used for the Trace Comparison technique.

The most important difference is that we could not perform the technique on the `ModeSelector` function block. Here, we had to cancel the process after 1 hour, since the model checker calls took too long ruling our candidates. In this case, the technique fails. Additionally, in the third `EmergencyStop` problem, the faulty code was almost hit, hence we put the 1 in parentheses.

In all other cases, the candidate exclusion technique was able to detected the faulty line while reducing the number of candidates to 1–8. Note that the number of hits is slightly better than the Trace Comparison technique, which did not find the error location in two instances. The number of potential candidates, however, is higher for the Candidate Exclusion technique.

7.5 DISCUSSION & COMPARISON

In summary, we conclude that neither the trace comparison nor the candidate exclusion technique distinguished themselves as being superior. Both techniques are suitable heuristics to narrow down the possible error locations and, sometimes, even able to find the exact error location. While the Candidate Exclusion technique sometimes cannot be applied since it takes too long, it did find the error location in all other cases. Its runtime and number of potential candidates, however, is higher than the Trace Comparison.

The Trace Comparison technique was tailored for the abstraction techniques detailed in Chap. 4 and is not readily transferable to other verification techniques.

Program	Sev.	Change	Loc	$\lvert\pi_c\rvert$	$\lvert C\rvert$	#Hit	Time
`EnableSwitch`	Simple	Ass.	120	15	6	1	< 1 s
`EnableSwitch`	Medium	Ass.	120	80	6	1	3 s
`EnableSwitch`	Simple	Add.	121	81	8	1	1 s
`EmergencyStop`	Medium	Ass.	115	67	2	1	1 s
`EmergencyStop`	Simple	Branch.	115	13	6	1	< 1 s
`EmergencyStop`	Complex	Miss.	111	16	1	(1)	< 1 s
`SafetyRequest`	Complex	Miss. & Ass.	140	193	3	2	3 s
`ModeSelector`	Simple	Ass.	155	26	—	—	∞
`GuardMonitoring`	Complex	Add.	110	17	6	1	3 s

Table 7: Case Study of the Candidate Exclusion technique

Its big advantage, on the other hand, it that this technique also generates possible corrections for the error.

By way of contrast, the candidate exclusion technique works without knowledge of the internals of the model checker or tweaking the state space generator and thus can easily applied to other abstraction techniques, verification algorithms or even model checkers.

7.6 RELATED WORK

Wong and Debroy [123] present a survey about different counterexample-based software fault localization techniques that have been studied for programming different languages. We present the most important works that relate to the techniques we described in this chapter.

Renieris and Reiss [102] also inspect nearest neighbors. They compile so-called program spectra, which represent information gathered during the execution of a trace. These spectra are then interpreted as binary vectors and compared using the Hamming distance. Groce et al. [65] describe a semi-automatic approach that also uses distance metrics to explain counterexamples. The also observe that there is no single best algorithm for fault localization because of the inherently subjective nature of the problem. Similar to our approach, Kumazawa and Tamai [76] then use the Levenshtein distance as a metric for comparison, which also gives the correction proposals. In contrast to our approach, they analyze infinite counterexample traces and liveness properties.

Sülflow and Drechsler [117] evaluated SAT-based techniques to locate errors in PLC programs written in IL. Their approach only consider the faulty trace and does not take information of the non-violating traces into account. They use techniques and correction-based debugging [113] to reduce the potential error locations for the user.

In this chapter, we presented and compared two approaches the for automatic error localization in counterexamples for PLC programs. As other authors, we used the Levenshtein distance for one technique, yet geared towards the cyclic scanning mode of PLC programs.

7.7 conclusion & future work

In this chapter, two heuristics were presented that allow to extract the possible cause of a violation of a property from a program. The key idea of these techniques is to compare violating and non-violating runs to gain knowledge about possible erroneous program locations. During our experiments and during our work on other model checking techniques, the techniques presented in this chapter helped tremendously to assess the validity of a counterexample. Due to their heuristic nature and the general problem to define what an exact cause of an error is—especially when an error is caused by missing code—the techniques cannot always succeed. Yet, the techniques were successful most of the time during our experiments and could reduce the potential error locations to a few candidates.

A limitation of the presented techniques is that they work purely on syntactic differences of whether a line is executed or not. Using semantic analyses [54] that take the actual values of the program variables into account, the accuracy could be further improved. This is especially so for programs with longs chains of instructions without branches.

8

STATIC ANALYSIS OF PLC PROGRAMS

In the previous chapters, we were concerned with verifying user-specified properties via model checking. The properties could either be provided as CTL formulae or automata. In this chapter, we will focus on inferring properties of PLC programs directly, without the need for user-supplied specifications. The goal is threefold: First, we want to provide a possibility to inspect possible values of variables. The user should, e. g., be able to inspect a succinct representation of the outputs of a PLC program, which can easily be checked for consistency. Second, we want to find potentially erroneous PLC code automatically. That is, we are looking for code that exhibits undefined[1] or implementation-defined (potentially ill-defined) behavior. This includes divisions by zero and out of bounds accesses of arrays, as well as *suspicious* constructs such as unreachable code or redundant assignments. Finally, we want to infer program properties that can speed up other analyses. If, e. g., we can infer a summary of the behavior of a function block, we can skip the evaluation of certain function block in the model checker. This approach will be further investigated in Chap. 9.

A key difference to the previous chapters is that we are no longer restrained to the observable behavior of the PLC. Since we are inspecting the behavior executed during the cycle, we also consider the intermediate instructions of the PLC.

8.1 APPROACH

Control Flow Graph

Our approach operates on the control flow graph (CFG) of the program. This graph contains all instructions of our intermediate representation. Two nodes are connected using a directed edge if they are connected w. r. t. the control flow of the program. For each node we then compute an over-approximation of all possible values that are stored in each variable during the execution of the controller. The analysis is performed in a flow-sensitive way based on the work of [44]. Function and function-block calls can optionally be handled in a context-sensitive way. The key idea of this static analysis is to use abstract interpretation to simulate the

1 By *undefined behavior* we subsume constructs that are either explicitly marked as undefined (such as accessing the *current result* after a function block call) or not clearly defined in the standard such as overflow of integer variables.

program not over a concrete domain such as integers, but over an abstract domain such as intervals. This is achieved by keeping track of the abstract values of each variable in each node of the CFG. Then, each edge of the CFG is abstractly executed and merged with the information of its successor nodes until the system stabilizes, i. e., all values have been seen.

To make the approach applicable to large programs comprising a large number of variables and program lines, we implemented two optimizations:

- We only track variables that are *visible* in the current context. That means that, e. g., we do not track the variables of the caller of a function block in the callee.
- We only track variables that are *live* at the current instruction. Live means that they are not over-written before being read again.

While the former optimization is based on a purely syntactic property, the latter is determined using a pre-analysis.

Once the static analysis has finished, we can present the information directly or we can further apply a set of checks to the computed variable ranges. For each division, e. g., we now check whether the divisor might be zero. We also check for correct indirect accesses, i. e., whether arrays and structured types are accessed using an index with correct bounds and correct types. Finally, our techniques allow for the checking of conditional expressions that are always evaluated to true or false as well as for unreachable code. Such a detailed analysis is currently not offered even by commercial PLC tools; an overview is given by [95].

8.1.1 *Contribution & Outline*

In Sect. 8.2, we detail our static analysis approach. This approach is based on abstract interpretation, with two crucial liveness based optimizations (Sect. 8.2.6 and Sect. 8.3) to reduce the size. Sect. 8.5 describes the checks we implemented on top of the static analysis results. In Sect. 8.6, we show the results of an industrial case study, for which we implemented a specific check for the controllers used there. We also checked our own implementation of the PLCopen safety function block library. We start by giving an overview of related work.

8.1.2 *Related Work*

To the best of our knowledge, Bornot et al. [33] were the first to describe a static analysis for PLCs. Their approach is also based on abstract interpretation, yet is limited to the interval domain and small Instruction List programs. Our work aims at verifying large scale PLC programs and therefore introduces abstractions to limit the scope of variables (cp. Sect. 8.2.6 and Sect. 8.3) and additional (bit-

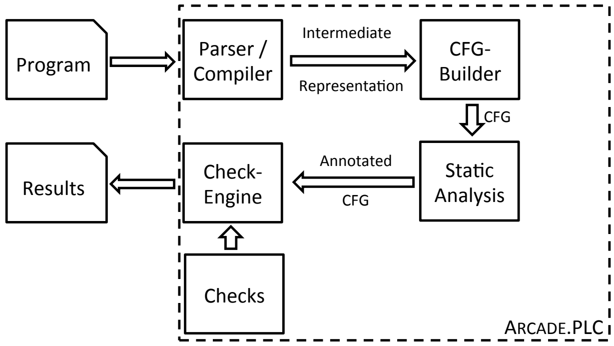


Figure 22: The static analysis process with ARCADE.PLC [28].

wise and value-set) domains. Both works are fundamentally based on the abstract interpretation framework described by Cousot and Cousot [44].

Chen et al. [37] describe a technique to make computing fixed points more efficient using so-called context projections. In their paper, *reachability* is examined as a special case of context projection. In this work we use a liveness based pre-analysis to make the analysis more efficient that follows a similar line of research.

Gourcuff et al. [63] examine abstractions for model checking PLC programs by taking the dependency of expressions and variables into account. In [62], they also verify Structure Text programs, yet their approach is limited to a subset of the language. They, e. g., do not allow for loops, while our approach supports all Structured Text features.

The techniques and results presented in this chapter were in part presented in previous publications. Part of the static analysis process in Sect. 8.2 is described in [115] and [28]. The latter also describes the localization of function block variables in Sect. 8.3. The ideas of the analysis for the correct usage of retain variables described in Sect. 8.4.1 was first published in [68]. Some ideas for the summarization of FBs described in Sect. 8.7 were published in [27]. The results of the case study given in Sect. 8.6.1 were first presented in [115].

8.2 STATIC ANALYSIS PROCESS

Our analysis process comprises four steps, which are depicted in Fig. 22:

1. We translate the PLC program into an intermediate representation (IR), as shown in Sect. 3.4, i. e., we operate on the same IR used for model checking.

2. We then create a control flow graph (CFG) out of the IR.

3. This CFG is then analyzed in a flow-sensitive way using an abstract interpretation framework, yielding an annotated CFG. In the annotated CFG, every

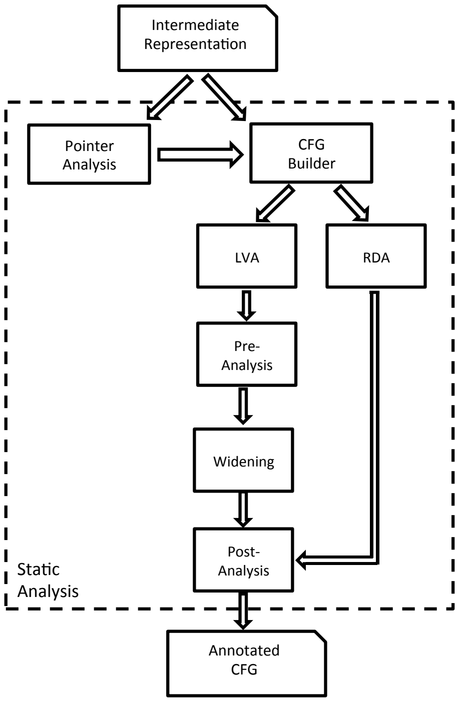


Figure 23: The detailed steps of the value-set analysis for PLC programs.

node contains an over-approximation of the values each variable can assume in this node.

4. Finally, we run a set of predefined checks on the annotated CFG and show possible warnings to the user. The user can also inspect the raw results of the analysis.

The abstraction interpretation step is further divided into detailed steps shown in Fig. 23. We explain these steps in the following.

8.2.1 *Pointer Analysis*

The pointer analysis infers for each pointer and reference of the program a list of potential pointees, i. e., variables it might point to during runtime. This analysis is run as the first static analysis step even before building the CFG. This is necessary, because the results of the pointer analysis are needed for the CFG builder to infer possible destinations of indirect function blocks calls[2]. For the analysis, we first gather all pointer variables and internal references (cp. Sect. 3.4.4).

2 While indirect function block calls are not defined in the standard, it is possible in some dialects to create an array of function blocks, and then call the i^{th} function block in this array.

```
FUNCTION_BLOCK SMALL_EXAMPLE
VAR_INPUT A, B: BOOL; END_VAR
VAR R : R_TRIG; END_VAR
VAR_OUTPUT OUT: INT; END_VAR

R(CLK := B);
IF NOT A OR R.Q THEN
        OUT := 1;
ELSIF NOT A AND R.Q THEN
        OUT := 2;
ELSE
        OUT := 3;
END_IF;
END_FUNCTION_BLOCK
```

Figure 24: The example program (extended version of [28]).

Then our analysis works in a control insensitive way (usually called *Andersen-style analysis* in literature, due to [1]). It inspects all `Assign`, `Alias`, `Member` and `Index` instructions of the program, thereby collecting all possible pointees on the right hand side and updating the left hand side accordingly. That is, each write to a pointer updates the set of its pointees. We merge it (or, if there is an indirect write, all of them) with the set on the right hand side. These steps are repeated until the system stabilizes.

8.2.2 *Control-Flow-Graph Builder*

After the pointer analysis, the CFG of the program is built. First, we build a graph for each POU. The nodes of this graph are the instructions of the IR (cp. Sect. 3.4.4). Two nodes n_0 and n_1 are connected using a directed edge $n_0 \rightarrow n_1$ if the control flow reaches from n_0 to n_1 (either because n_1 is direct successor of n_0 or there is a jump from n_0 to n_1). Multiple successors can only arise from conditional jumps. In this case, edges are labeled with constraints covering the conditions according to the conditional branch instructions.

In the next step, call edges are added for all `CALL` instructions to other POUs, resulting in a so-called *super graph* of the program. We add return edges from the exit node of the called POU to a *return node* in the CFG. The return node will later gather the effect of the call. We support two different ways of handling the granularity of the analysis of calls: First we can analyze a POU in a context-insensitive way. This means that the code for a function or a function block instance will only appear once. Multiple calls to this instance will then result in multiple edges to the same entry. Consequently, return edges will be generated to all return nodes, which allows the entrance of the POU from one call site, but the exit to a different call site, effectively over-approximating the possible behavior. While

Super Graph

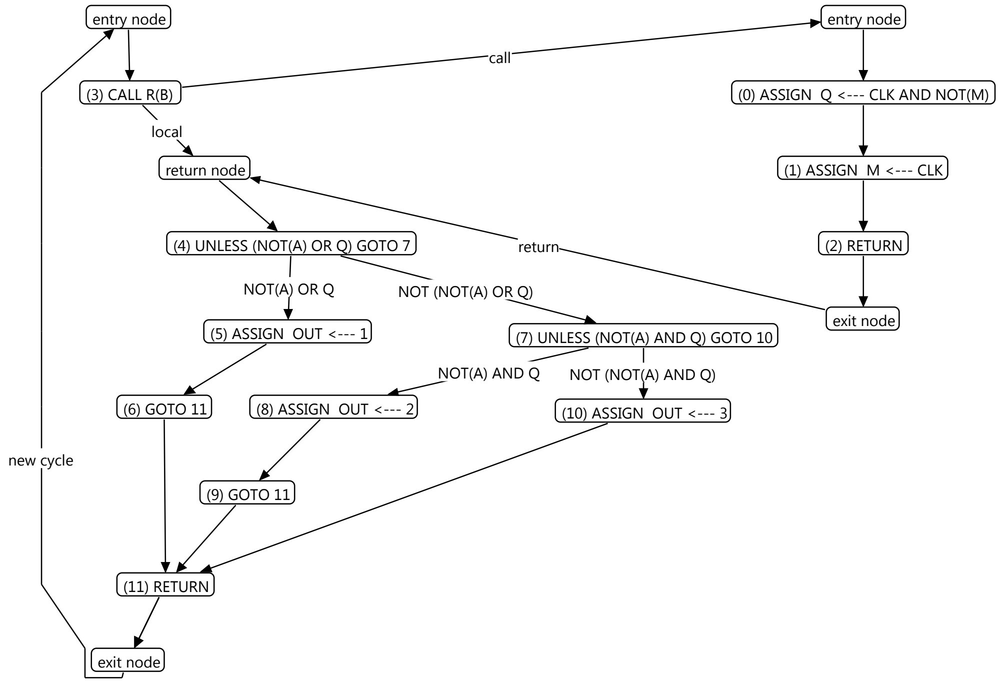


Figure 25: The CFG of the example program showing the internal representation. Nodes with actual instructions have a unique label, which is used to associate analysis information.

this is generally faster because less code has to be analyzed, we also allow for analyzing POUs in a context-sensitive way. To enable a context-sensitive analysis, a POU declaration can be marked with a special pragma (cp. Sect. 3.4.2). Then, the CFG builder will generate a new copy of the CFG of the POU for each invocation of a POU instance. Most standard function blocks in our implementation are already marked, since their body is quite short, whilst they still benefit highly from a context-sensitive analysis.

Context-Sensitive Analysis

Worked Example

In the following, we exemplify all steps of the algorithms using the example ST program shown in Fig. 24, which is an extended version of the program used in [28]. This program performs the following operation on the inputs `A` and `B` in every cycle: First, an `R_TRIG` function block instance `R` is called to detect a rising edge on `B` (signaled in `R.Q`). Then, if `NOT A` or `R.Q` is set then `OUT` is set to 1; if otherwise `NOT A` and `R.Q` is set then `OUT` is set to 2; if neither applies then `OUT` is set to 3. The corresponding CFG of this program is shown in Fig. 25. In our IR, both `IF`s statements are translated into conditional branch statements with the corresponding constraints written in the outgoing edges. The scanning cycle is indicated by the left-most edge. The instance of the `R_TRIG` function block is shown in the right.

8.2.3 *Static Analyses Dataflow Framework*

We use a data-flow framework to implement the further flow-sensitive analyses on the CFG. Such a framework is defined over a lattice $(L, \sqsubseteq, \bot, \top)$ (cp. Sect. 3.3.1), i. e., a po-set w. r. t. $\sqsubseteq$. We define two analysis-dependent operations for states $S \subseteq L$:

$$\begin{aligned} \text{transform operation for } op \quad & f^{op} : S \to S \\ \text{join operation} \quad & \sqcup : S \times S \to S \end{aligned} \tag{1}$$

The transform operation f^{op} captures the semantics of an operation op on a set of states S. The join operation monotonically merges the information of multiple states into one state. For each node n in the CFG, we now associate a state S_n to n, which is initially $\bot$. We build data-flow equations on the CFG between these nodes using the transform and join operation:

Data-Flow Equations

$$\begin{aligned} S_n^{\text{out}} &:= f^{op}(S_n) && \text{where } op \text{ is the operation of } n \\ S_n &:= \sqcup_{i \in \text{Pred}(n)} S_i^{\text{out}} && \text{where } \text{Pred}(n) \text{ are the predecessors of } n \end{aligned}$$

These equations are evaluated until the system becomes stable. Observe that the monotonicity of the join operators ensures that always $S \sqsubseteq S'$. Termination, however, still has to be carefully evaluated if L admits infinitely ascending chains.

Work List Algorithm

To efficiently compute a fixed-point, we perform the analysis using a work-list algorithm. In the first step, we put all successors edges of the entry node into the work-list. Then, the following steps are performed:

1. If the work-list is empty: Stop.
2. Get edge e out of the work-list. Suppose e connects nodes n_0 and n_1, S_{n_0} and S_{n_1} are their associated states and the instruction of n_0 is *op*.
3. Perform transform operation $S_{n_0}^{\text{out}} := f^{op}(S_{n_0})$.
4. Merge $S_{n_0}^{\text{out}}$ and S_{n_1} resulting in $S_{n_1}^{\text{in}} := S_{n_0}^{\text{out}} \sqcup S_{n_1}$.
5. Unless $S_{n_1}^{\text{in}} \sqsubseteq S_{n_1}$ put the successor edges of n_1 into work-list.
6. Associate $S_{n_1}^{\text{in}}$ with node n_1.
7. Goto step 1.

We organize the work-list in way such that the edges are processed in reverse postorder by implementing the work-list as a priority queue where the offset of the node in the translation unit is the priority. This is not crucial for correctness but ensures that the system stabilizes faster by first computing earlier nodes in the CFG.

This framework can be applied to either analyzing a single program or a whole controller, where multiple POUs interact using shared global variables (cp. Fig. 4 on p. 15). If we are analyzing a single program, we assume that all global variables have an unknown value. If, on the other hand, we are analyzing a whole controller, we set the global variables to their default value, which allows us to analyze the interaction of global variables between programs.

8.2.4 *Live Variable & Reaching Definition Analysis*

Live Variable Analysis (LVA) is a data flow analysis that determines the set of *live* variables for each node in the CFG (cp. [109], Chap. 1.15).

Live Variable

Definition 8.1: A variable is called *live* if its value is read somewhere in a successor node (or, in other words, if its value is not overwritten on all successor paths before it is read). We call a variable *dead* iff it is not live.

Using our framework this analysis is implemented by traversing the CFG in reverse order (i. e., by reversing the direction of all edges and starting in the exit node). The lattice is $\wp(2^{\text{Var}})$, i. e., each state is a bitset where each live variable is marked. $\perp$ denotes that all variables are dead. For each variable v, we define further:

$$f^{op}(S)(v) = \begin{cases} 0, & \text{if } v \text{ is unconditionally written but not read by } op \\ 1, & \text{if } v \text{ is read, or read and written by } op \\ S(v), & \text{otherwise} \end{cases}$$

The join operation defines a variable live if it is live in either state:

$$(S_0 \sqcup S_1)(v) := \max(S_0(v), S_1(v))$$

LVA Example

In the example CFG in Fig. 25, we have `A` live in nodes $\{0,1,2,3,4,5,7\}$ since they are no longer needed after these nodes, and are reset at the beginning of the next cycle. Similarly, variable `B` is only live in node 3, because it is not read after the call. `R.Q` become live in node 1 and stays live in nodes $\{1,2,4,5,7\}$. Finally, `Out` is live in nodes $\{2,5,7\}$ since it is an output variable and thus relevant at the end of the cycle. It is dead at the other nodes since it is overwritten before being read again.

LVA is a backward analysis, i. e., the CFG is processed in reverse order. A variable is set live, if its value is used (i. e., it is read / it appears on the right hand side of an expression) and a variable is set dead, if its value is overwritten (it appears on the left hand side of an assignment).

Reaching Definition Analysis

Complementary to the LVA, the Reaching Definition Analysis (RDA) computes a list of nodes for every variable in every node. This list indicates all possible places where the variable has been defined previously.

Reaching Definitions

Definition 8.2: Let n be a node in the CFG and v be a variable. A *reaching definition of v in n* is a node d in the CFG, with:

- v is written in d, and
- there is a path $d \to n_0 \to \cdots \to n_i \to n$ and v is not unconditionally overwritten in $n_0 \ldots n_i$.

The RDA is implemented using our framework by traversing the CFG in forward order. For each variable, the lattice is $\wp(2^{\text{NODES}})$, i. e., each state is a bitset where the reaching definitions are marked. $\perp$ denotes that no definitions are reachable. For each variable v, we define the following data-flow equations:

$$f^{op}(S)(v) = \begin{cases} \{n\}, & \text{if } v \text{ is unconditionally written in } n \text{ by } op \\ S(v) \cup \{n\}, & \text{if } v \text{ is conditionally written in } n \text{ by } op \\ S(v), & \text{otherwise} \end{cases}$$

The join operation merges the reaching definitions:

$$(S_0 \sqcup S_1)(v) := (S_0 \cup S_1)(v)$$

RDA Example Continuing the example shown in Fig. 25, the reaching definitions of `Output` in node 11 are nodes $\{5, 8, 10\}$.

8.2.5 *Value-Set Analysis*

We now present the core analysis. For every node of the CFG, the value-set analysis (VSA) determines an over-approximation of the possible values each variable can assume in this node. This information is the basis for all further analysis, and builds on the pointer analysis (and—as an optimization—on the LVA and RDA analyses results).

Pre-Analysis To speed up the VSA, we first perform a pre-analysis. In this pre-analysis, we determine the set of variables that are not aliased by a pointer and are syntactically constant, i. e., never written during the runtime. For this, we check the points-to sets of all pointers determined in the pointer analysis. Then, we iterate over all instructions in the CFG that perform an assign or a call. If a variable never appears on the left hand side of an assign, and also not as an output parameter of a call, it can be assumed constant. Hence, these variables do not have to be tracked during the VSA. Depending on the dialect and which style the PLC program was written in, it may be that a great number of variables are used as constants, and thus that the total number of variables can be reduced.

Data-Flow Equations We then perform an abstraction interpretation using our data-flow framework and our abstract domains defined in Sect. 3.3. Each transfer function f^{op} and the join operator $\sqcup$ is defined according to the domains. Since such a system converges very slowly for intervals or might even diverge with an infinite chain of increasing interval bounds, careful considerations have to be taken to assure termination. We defer a solution for this to Sect. 8.2.7.

Conditional Branches The analysis presented thus far already results in a valid over-approximation. We do not, however, consider the conditional branch instructions, which manifest themselves as constraints on the edges of the CFG. Since these constraints provide valuable information to make the analysis more precise (by restricting the set of possible values in the current branch), we want to incorporate them. This can be done by intersecting the current abstract state with the branching constraint using the $\sqcap$ operator. For this, we use the constraint solver presented in Sect. 4.3 to generate reachable intervals for simple arithmetic and Boolean constraints. A more systematic way to perform this using a SAT-based refined scheme has been presented by us in the past [17]. Yet, for efficiency reasons, we use the hand-written constraint solver.

Context-Sensitive Analysis Note that the value-set analysis works on the CFG that already incorporates edges for call instructions to other POUs. The decision as to whether the analysis is performed in a context-sensitive or context-insensitive manner is thus decided by our CFG builder.

Worked Example We continue our example from Fig. 25. The VSA will start at the entry node, setting the inputs `A`, `B` to $\top$, the output `OUT` to $\bot$. The value of `B` is passed to the `R_-TRIG` block, setting its `CLK`, `M` and `Q` to $\top$. We obtain in node 5: `OUT` $= 1$, `A` $=$ `Q` $= \top$ (this cannot be represented more exactly using our domains) and in node 7: `A` $= 1$, `Q` $= 0$. We then get in node 8: `OUT` $= 2$, `A` $=$ `Q` $= \bot$ and in node 10: `OUT` $= 3$, `A` $= 1$, `Q` $= 0$. Finally, in node 11 we merge the results to `OUT` $= [1, 3]$, `A` $=$ `Q` $= \top$. After the next iteration the system stabilizes.

8.2.6 *Value-Set Analysis with Sparse Memory States*

Thus far, we computed the abstract value for each variable in each node in the CFG. This is potentially wasteful and can seriously impact the applicability of the technique to real programs, especially if the program contains a huge number of variables only relevant to parts of the program. To alleviate this problem, we turn to an abstraction that only keeps track of a part of the variables.

Once we have the liveness information in each node, we can make use of the liveness information to select the relevant variables. The key idea here being that it is not necessary for us to have to store the abstract values of dead variables, since their values are not used in the future (either because they are not used at all, or they are overwritten before being used again). In practice, this results in much smaller abstract states. Even for our small example program, the number of variables that have to be stored in each abstract state would be roughly half the amount, i. e., it is not necessary to store the value of `Out` in nodes $\{0, \ldots, 7\}$ and it is only necessary to store the value of `A` in nodes $\{0, 1, 2, 3, 4, 7\}$.

8.2.7 *Widening*

Crucial for the termination of the data-flow algorithm in Sect. 8.2.3, is step (5), which puts the successor edges of n_1 into the work-list unless $S_{n_1}^{\text{in}} \sqsubseteq S_{n_1}$. If $S_{n_1}^{\text{in}}$ only slightly increases each iteration, the convergence might be too slow in practice. If the domain admits infinite ascending chains, it might even fail to terminate. Therefore, we use *widening* to accelerate this process. If an edge in the CFG is analyzed more times than a certain threshold, we activate widening for this edge. During widening, we directly saturate our abstract value while merging two states. We use 5 as the default threshold, but allow for a reconfiguration. We selected 5 because it turned out to be a good middle ground between runtime and precision, especially when sets of discrete values (e. g., enumeration values, diagnosis codes, etc.) are summarized using our value-set domain. Widening operators have been extensively researched in the past, see [43] for an overview. We implement special widening for intervals only, all other domains are directly widened to $\top$. For intervals, we first set the increasing interval bound to $\pm\infty$.

To illustrate the widening process with intervals, suppose a program that is incrementing the variable `x` by 1 in a loop. Suppose further that we have at the start of the loop $x \in [1, 10]$ and thus $x' \in [2, 11]$ at the end of the loop. For the next iteration, we now merge x and x'. That is, we compute $x'' = x \sqcup x' = [1, 11]$. Once we activate widening, we compute $x'' = x \nabla x' = [1, \infty]$, where ∇ denotes the widening operator. Widening thus sets the upper interval bound directly to ∞.

In practice, the widening step is crucial to make analyzing PLC programs with for-loops and counters possible, since the static analysis algorithm would otherwise take too long to find a fixed point. A simple counter implemented using the `DINT` type, for example, would require 2^{32} steps to converge without widening, while the approach using widening converges after the threshold of 5 iterations.

8.2.8 *Post-Analysis*

In the last step, we perform a post-analysis to produce the annotated CFG. We cannot directly use the results from the values-set analysis, since we do not store the values of all variables in each node but only the live variables. In our worked example, e. g., the variable `Out` is not live in node 8. It is not live, because its value is unconditionally overwritten and its new value does not depend on the old value. While `Out` becomes live after the assignment, we do not have its value right before the assignment available. Yet, we want to annotate `Out` with its old value to enable certain warnings. If, e. g., `Out` already contains the value that is written to it, the assignment has no effect and might thus be erroneous. Hence, we want to annotate all variables that appear in each node of the CFG with their values. To do so, we first check whether the values are stored in the value-set information of this node. This is the case if the variable is live. Otherwise, we make use of the RDA information: To reconstruct the old value of variables that are not live, we read the value in all possible reaching definitions and merge the values. We obtain an annotated CFG where all variables in each are annotated with an over-approximation of the values that are possible in this node.

8.3 LOCALIZATION OF FUNCTION BLOCK VARIABLES

We use the LVA to compute the liveness of variables of the program. This liveness, however, is computed in a scope-agnostic way, i. e., a variable that is live at the end of a function call (because its value is needed in the next call of this function) will be live in the caller's scope. The result of this is that internal variables of function blocks—even if they cannot be accessed from outside—are live everywhere. Tracking the values of these variables in the whole CFG is wasteful if they are only accessed inside the function block.

Local Variables

To exemplify, such a situation is depicted in Fig. 26 (1). Suppose that the variable a is only accessed in the function block FB (indicated by one read and one write). Due to the read, it is live at the call statement to the FB call[3], making it live in the calling program, effectively becoming live almost everywhere. This is indicated as solid lines in the figure, while the dotted line indicates the part where it is dead between the read and the write. Note that this happens, although it is never accessed in the calling program. It would also happen if it was not visible in the calling program.

To alleviate this situation, we introduce another technique. The key idea being that there is no need to not propagate liveness information of local variables through call edges. The result is shown in Fig. 26 (2). The local variable a is live at the beginning of the FB (due to the read), but the liveness is not propagated through the call edge, effectively making it dead in the caller's scope. Observe that while it is dead at the return edge, it is live from the write to the end of the function. When performing the value-set analysis, we have to add a data flow edge from the end of the function to its start. We then propagate the results from the exit node of the FB directly to the entry node. The result is that we do not have to track the value of a in the main program.

Thus far, we considered variables that cannot be accessed from outside. Input and output variables, however, can also be accessed from the caller's scope, cp. Sect. 2.2.7. In the following, we discuss how we handle these cases.

Output Variables

If a is not a local variable but an output, its value can be read outside of the FB. Suppose it is read after the call, as depicted in Fig. 26 (3). Then, a is live at the read outside the FB. Since liveness propagates backwards, a becomes live at the return statement as well. The important aspect here is that although a is accessed outside the FB, it is still not live everywhere in the caller's context.

Input Variables

Finally, suppose a is an input variable. Then, a can be written outside the FB as shown in Fig. 26 (4). We distinguish between the two different ways to pass

3 Recall that liveness is propagated backwards.

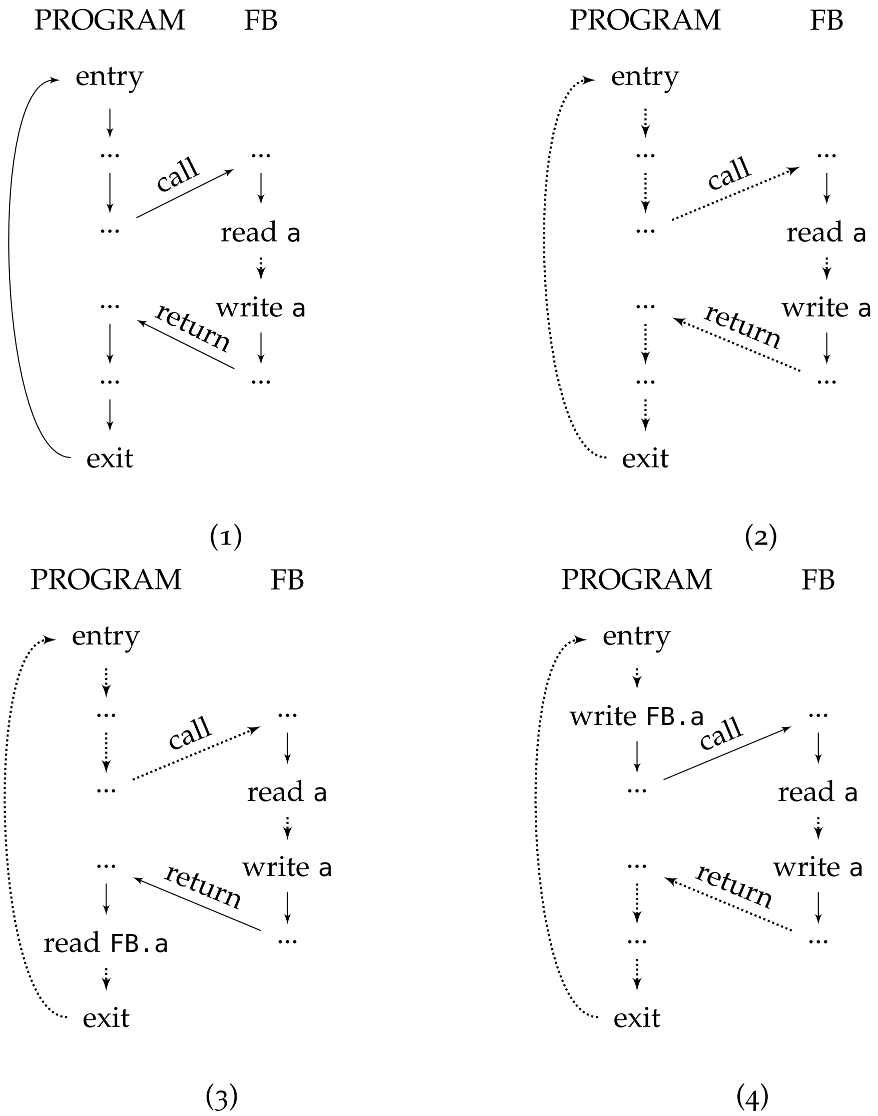


Figure 26: Localization strategy for FB variables. Edges where a is live are drawn as solid lines, edges where a is dead are dotted. (1) naïve approach, (2) scoping of LVA, (3) reading a outside of its scope, (4) writing a outside of its scope [28].

```
1  PROGRAM Program
2  VAR (* RETAIN *)
3    first_cycle : bool := true;
4  END_VAR
5  VAR
6    a : int := 0;
7    b : int := 1;
8    c : int := 0;
9  END_VAR
10   if first_cycle then
11      (* initialization block *)
12      c := 2;
13   end_if;
14   first_cycle := false;
15   a := b / c;
16 END_PROGRAM
```

Figure 27: PLC program with initialization [68].

parameters to FBs (cp. Sect. 2.2.7): First, the value of a can be passed as a (named or unnamed) parameter in the call statement as an argument. In this case, a is overwritten in the call-edge and thus not live before the call edge. If, however, a is not passed in the call, the previous value (possible set in the caller context) is relevant, and thus a should be live at the call site.

Applying these checks to our running example, we get the following results: Since we have in node 7: A=1 and R.Q=0, we can infer that `NOT A AND R.Q` is always evaluated to false. Additionally, it is impossible to take edge $7 \rightarrow 8$ and thus, we can warn about dead code in node 8. We can hence detect this logic error in the program, map the node to its location in the source file and present the error to the user.

8.4 INITIALIZATIONS & PARTIAL UNROLLING

In a PLC program it is sometimes necessary to run initialization code only in the first scan cycle. The program in Fig. 27, e. g., uses the variable `first_cycle` to control the execution of the initialization block. Therefore, the program initializes the variable `first_cycle` to `true` and sets it to `false` in all later cycles.

Observe that the initial value for the variable c is 0, but it is directly set to 2 in the initialization block. Hence, there is no division by zero in line 15. The value set analysis, however, would infer that c can be $\{0, 2\}$ at the start of the program and because `first_cycle` can be {`true`, `false`} it would generate a spurious division-by-zero warning.

To avoid such warnings, we unroll the program once. That is, we analyze the program once without considering the PLC cycle. Then, we use the value-set information of all variables at the end of the program to re-analyze the program, now considering the PLC cycle. In the example, we would infer that `first_cycle` is always `false` and `c` is always 2 after the first cycle, thus avoiding the spurious warning.

8.4.1 *Retain Variables*

As we have seen in Sect. 2.2.4, variables can retain their value between restarts of the PLC. Effectively, this means that the program is restarted with all variables reset to their initial value, while some variables marked as `RETAIN` or `PERSISTENT` still contain their previous value. On the one hand, this makes the partial unrolling technique unsound, since we do not consider these new initial values (without the unrolling technique, however, our approach is sound, since we consider all possible values for all variables). On the other hand, inducing new behavior *after* a restart of the PLC via the use of retain variables is most likely not the indented behavior of the programmer. Hence, we want to warn about these situations.

Detecting Unwanted Behavior of Retain Variables

To detect additional behavior induced by retain variables, we implemented the following analysis [68]: After the first unrolling and analysis of the program, we start a second analysis. This time, we keep the computed value sets of the retain variables and reset all other variables to their initial values. We again perform an unrolling step and then analyze the cyclic behavior of the program. Now we have two annotated CFGs: one from the first analysis and one from the analysis after the first restart of the PLC. By comparing the value-sets in both CFG, we can detect all nodes where new behavior can arise due to a restart. We issue a warning for these cases.

We merge the values of both CFGs and use the result as the annotated CFG on which all further checks are implemented on. This makes the analysis sound again, even with the unrolling technique.

8.5 implementation of checks

Once the static analysis returns an annotated CFG we run our checks, which are detailed in the following section. The checks are implemented on top of a framework that offers to inspect different categories of the program, depending on the granularity and type of object they have to check:

- Checks can inspect certain instructions. To check whether the index of an array is out of bounds, we only have to inspect all `Index` instructions.
- Checks can inspect the expressions in all instructions filtering out certain operations.
- Checks can inspect the global summary of the variables.

Using this framework, we implemented the following checks:

Division by zero

For every division expression in the program, we check whether the number 0 is contained in the computed values of the divisor and warn, if this is possible. This value can be obtained from the variable annotation in the CFG node containing the division operation.

Overflow

For each assignment, this check verifies that all values on the right hand side expression fit into type of the left hand side without overflowing. That is, we test whether the set of values of the right hand side is a subset of the values that the type of the right hand side allows. We are thus able to detect possible overflows. This check works for all `Assign` instructions and all assignments to the parameters for `Call` instructions.

Array index out-of-bounds

For each array access, we check whether the result of the index expression is within the bounds of the array. Otherwise, we issue a warning that includes the interval the index variable might lie in, such that the developer can check the legitimacy of this warning.

Constant Variables

We check for variables that only contain a constant value during execution. This either indicates a stylistic issue, in which case the variables should be declared as constants. Alternatively, it might indicate a problem in the program.

Constant Written Variables

This check is an extension of the previous one. First, we analyze which variables are written to in the program. We then issue this warning for all variables that are written to and still have a constant value during execution. While the previous check usually only indicates a stylistic issue (all variables that contain a constant should be marked as a constant), this warning indicates a real problem: A variable does not change its value during runtime although the program writes to it.

Missing Case Labels

For each `case` statement, we check for missing case labels. That is, if a `case` statement has no `else` clause, we check that each value in the case expression is handled. Otherwise, we issue a warning that contains the values that are not handled.

Unreachable Code

We mark the beginning of each position in the code that is unreachable.

Condition always true/false

For each Boolean condition that determines conditional control flow in the program, we check whether the outcome can be true or false. If this is not the case, i. e., the variables are too constrained to allow for different outcomes, we issue a warning.

Partial condition always true/false

We check for each Boolean expression in the program whether the outcome can be true or false. Note that in contrast to the previous check this one also checks partial expressions. It is therefore more sensitive than the previous check.

Redundant assignment

If we can prove that the right and left hand side of an assignment always contain the same (single) value, i. e., a value is stored into a variable that is already stored there (in every context), then we issue a warning.

Assignment might loose precision

We issue a warning if the interval computed for the right hand side of an assignment of a variable, or a parameter, does not fit into the type bounds of the assignee.

Possible violation of assertion

We defined an `ASSERT` function in a dedicated `ARCADE` namespace. This function takes a `BOOL` input value and has an empty implementation. If we cannot statically prove that the input value is true, we issue a warning. This allows the user to manually insert many different kinds of checks into the code.

Possible violation user defined check

We issue a warning for each invocation of a POU instance where we cannot prove an annotated pre-condition (cp. Sect. 3.4.2).

Possible new behavior caused by retain variables

As described in Sect. 8.4.1, we issue a warning if a reanalysis after a PLC restart permits new behavior (using retain variables) compared to a normal start. The warning can generate false positives if a programmer is, e. g., counting the number of restarts. It detects, however, many situations where retain variables are used in an inconsistent way.

Program	#loc	#FBs	time	$\#W_1$	$\#W_2$	#FP
`App1 / Program1`	233	3	< 1 s	6	0	0
`App2 / Program2`	2776	100	11 s	0	8	0
`App2 / Program3`	169	5	3 s	0	0	0
`App2 / Program4`	2684	100	146 s	0	301	0
`App2 / Program5`	206	12	< 1 s	0	0	0
`App3 / Program6`	344	12	< 1 s	3	0	0
`App4 / Program7`	3339	18	40 s	9	50	9

Table 8: Part of the case study with anonymized program names [115].

Output written multiple times

It is usually good programming practice that each output variable is written at most once per cycle. This check verifies that an output variable will not be written multiple times per cycle. It is performed by checking at each write access of an output variable that the reaching definitions are still empty. If not, an error is presented that contains the previous write location.

8.6 CASE STUDIES

We evaluated the static analysis in various projects. In the following, we present the results of an industrial case study and the results verifying our own PLCOPEN library implementation.

8.6.1 *Industrial Programs*

Industrial Project

In an industrial cooperation, we checked a real-world project written for the ABB Compact Control Builder AC800M [28, 115]. This project comprises about 20 so-called *applications*, which are different programs interacting using global variables. In each application about 1000 global variables were used. In total, the project contains more than 100 programs, with about 50 000 lines ST code. In the programs, up to 100 function blocks were used. Each POU contains between 100 and 3500 lines of ST code. We could finish the static analysis, including all of our checks on all programs, in about 10 minutes. Without the LVA based optimization technique, we were not able to finish the static analysis. The high number of global variables and FB variables required in this project meant it was too costly to store the information in every CFG node.

Results

We selected some representative programs of the case study and anonymized their names. The results are shown in Tab. 8. The table shows the program we checked, the number of lines of ST code in the program (not including the functions and function blocks used in the program), the number of function blocks used (#FBs), the time for running the static analysis, the number of warnings in the program (#W_1) and the number of warnings in other organization units (#W_2), e. g., the function blocks used in the program.

Our checks can trigger in every location of the program, including the function blocks that are used in the program. This, however, results in a number of false positives for some warnings (summarized in #W_2). The reason for this is that the function blocks provide many extra functions, which are not necessarily used in the main program. To give a concrete example, a function block might have an input `Enable` to control the activation of some function. If the main program always enables this functionality, this input is hard-wired to `true` in every call. This then results that the warning *condition is always true* at the corresponding `IF Enable THEN` statement in the function block is generated. Therefore, we disabled these warnings for the function blocks and only activated them in the main program. After this change, the remaining warnings were mostly stylistic warnings about variables that could be declared as constants and redundant compares (#W_1).

8.6.2 *Specific Warning: Illegal GetStructComponent / PutStructComponent*

Programs written for the AC800M PLC can make use of special firmware functions called `GetStructComponent` and `PutStructComponent`. Using these functions the n^{th} component of a structured data type can be accessed, which is needed to support array-like data structures[4]. For every access, n must be greater than 0 or less than the number of elements in the struct, otherwise an error is signaled. Additionally, it is detected if the accessed element is of the wrong type. These are, however, runtime checks that are not prevented or detected at compile time.

A wrong program can thus fail at runtime, which motivates new warnings for the offline checking of correct usage of the functions `GetStructComponent` and the corresponding `PutStructComponent`. For this specific case study, we hence implemented the following checks:

- For each call of `GetStructComponent` and `PutStructComponent` we infer the interval for the index expression. We then check whether there are actually elements in the structure for all values in this interval. If not, we issue a warning that the structure might be accessed outside of its bounds.
- Additionally, we check that all elements of the structure with index in the inferred range have the same type. The rationale is that if they have a different type, the call to `GetStructComponent` or `PutStructComponent` might fail at runtime. We issue a warning if the type check fails.

```
CASE DiagCode OF
(* ... *)
16#8000:
    IF NOT S_ChannelNC OR S_ChannelNO THEN
        DiagCode := 16#8005;
        T_1(IN:=1, PT:=DiscrepancyTime);
    ELSIF NOT S_ChannelNC AND S_ChannelNO THEN
        DiagCode := 16#8001;
    END_IF;
END_CASE;
```

unreachable

Figure 28: Code fragment of `SF_Antivalent` implementation showing unreachable code caused by an erroneous specification.

8.6.3 PLCopen *Safety Function Blocks*

We also checked our own PLCopen safety function block library. All blocks could be checked in seconds. Most warnings were stylistic: Variables were tested twice and some sub-expressions in `IF` statements were always `true/false`. These warnings were harmless and usually caused by a literal copy of the respective expressions from the PLCopen standard. Redundant compares, e. g., are often used because the specification is written in a conservative way, i. e., input signals are tested again although they were already checked in a higher priority transition. One warning, however, caught our attention. This warning indicated *unreachable code* and is shown in Fig. 28. It was raised in the `SF_Antivalent` block and is caused by the same problem that we discussed in Sect. 6.5.2 (cp. Fig. 18 on p. 83). We can hence detect this problem either using static analysis or automata-based model checking.

8.7 CALCULATION OF SUMMARIES

The value-set analysis computes an over-approximation of the set of values each variable can take at each program location. Thus far, this information was used to produce warnings for problematic code. We can also use this information to summarize the behavior of programs or function blocks. This summary can then be presented to the user or be used for further analyses [27].

We implemented two summaries. The first one merges the values of each variable in all nodes of the CFG. It hence gives an overview of the ranges each variable resides in during the cycle. The second summary takes only the visible behavior into account and hence only summarizes the values of each variable at the beginning and end of every cycle (by taking only the input and exit nodes into account).

4 The `Array` type is missing on this platform.

We use the results of this analysis in Chap. 9 to speed up the model checking process.

This summary takes the different abstract domains (cp. Sect. 3.3) into account and thus provides a succinct representation of the possible values. To exemplify, a typical output of the summary of the visible behavior for the `SF_Antivalent` function block would look as follows:

- Output `Ready`: {`false`, `true`}
- Output `S_AntivalentOut`: {`false`, `true`}
- Output `Error`: {`false`, `true`}
- Output `DiagCode`:
  - [`0`, `16#C003`]
  - ⟨∗∗`000000000`∗`0`∗∗∗⟩
  - {`0, 16#8000, 16#8001, 16#8004, 16#8005, 16#8014, 16#C001, 16#C002, 16#C003`}

That is, the outputs `Ready`, `S_AntivalentOut` and `Error` might all assume the values `true` and `false`. If one of these outputs was, e. g., stuck-to-zero this problem would immediately be obvious to the developer. For the output `DiagCode` we get a list of possible values in different representations: Firstly, the value is represented as the interval [`0`, `16#C003`][5]. Then, the bitwise representation is shown. Finally, the value is represented as a set of distinct values. The latter representation is the most suitable of this variable type. A missing value or a wrongly coded value would immediately become obvious to the developer. If, e. g., one would assign `8004` to `DiagCode` instead of `16#8004` (i. e., a missing hexadecimal specifier, so the value is decimal) then the value `16#1f44` would appear in the list, thereby making the mistake obvious. Especially when developing function blocks, this helps tremendously in catching bugs early.

8.8 CONCLUSION & FUTURE WORK

In this chapter, we detailed how we implemented an efficient static analysis for PLC code. The core of this analysis computes an over-approximation of the values for each variable in each program location. This information can then be presented to the user, used in further checks, or stored for further analyses. Crucial for the efficiency of the analysis is the LVA information that allows to reduce the abstract program states to manageable sizes. With these techniques, we could apply our approach to large industrial programs.

5 The prefix `16#..` indicates hexadecimal constants

Using the checks we implemented, we were able to automatically detect many real world bugs in PLC programs, while at the same time having a very low number of false positives. We were also able to implement PLC specific checks, which can detect the misuse of certain firmware functions or inconsistent use of retain variables.

In contrast to the approaches using model checking, which we presented in the previous chapters, the static analysis works as a *push button* technique, i. e., it can be used without any manual effort by the user, especially without any effort in formulating the specifications. In practice, this means that many program errors can be detected without any additional costs, which makes the static analysis very attractive from the user's point of view.

The current drawback of the analysis lies in the domains we provided. Since we only implement non-relational domains such as intervals, relations between variables are not captured precisely. Sometimes, this can cause false positives, as in the following program fragment:

```
1 VAR
2        A : ARRAY[0..5] OF INT;
3        I, J: INT;
4 END_VAR
5 J:=0;
6 FOR I := 0 TO 5 DO
7        A[J] := 0;
8        J := J + 1;
9 END_FOR;
```

Here, we can infer that $\texttt{I} \in [0, 5]$, but not that $\texttt{I} = \texttt{J}$. Hence, we get a spurious warning in line 7 since we cannot infer the correct interval for J. In the future, (weak) relational domains such as convex polyhedra [46], two variables per inequality [111], or octagons [87] can be implemented for such cases so as to further reduce the number of remaining false positives.

9

STATIC ANALYSIS & MODEL CHECKING INTERPLAY

We implemented model checking techniques to verify specific program properties defined by the user, either by formulating CTL expressions or by safety automata. We also implemented static analyses that work without the need for specification by performing pre-defined checks or by computing program summaries. In this chapter, we will show how these technique can interact, allowing more efficient abstractions to verify larger programs. This will finally allow us to verify the safety application introduced in the beginning (Fig. 1 on p. 2). On the other hand, we will also explore techniques to improve the static analysis results using the model checker.

The idea of using static analysis for state space reductions has also been applied to verifying microcontroller binary code [104].

9.1 VERIFICATION OF A SAFETY APPLICATION

We now come back to the safety application shown at the beginning in Fig. 1 on p. 2. We want to verify that whenever either the emergency stop button is activated (using one of the redundant sensors) or the light curtain is triggered, the safe stop functionality is activated, which will then ensure that the motor eventually stops. Putting all of these requirements into a single formula is too complex, so we break it down into sub-problems. First, we want check that whenever the emergency stop buttons signals `S1_S_EStopIn_1` or `S1_S_EStopIn_2` become `false`[1] then the safety stop is activated:

$$AG\,\big((\neg \texttt{S1\_S\_EStopIn\_1} \lor \neg \texttt{S1\_S\_EStopIn\_2}) \\ \implies \neg \texttt{SF\_SafeStop1\_1.S\_StopIn}\big) \tag{1}$$

Similar, we want to verify this property for the light curtain:

$$AG\,\big(\neg \texttt{S2\_S\_ESPE\_In\_1} \implies \neg \texttt{SF\_SafeStop1\_1.S\_StopIn}\big) \tag{2}$$

1 These signals are implemented using reverse logic, so `false` means *stop requested*.

Finally, we want to verify that the safe stop SFB correctly responds and the motor eventually comes to a stop:

$$\begin{aligned} AG\ ((&\neg \texttt{SF\_SafeStop1\_1.S\_StopIn} \\ &\land \texttt{Internal\_Acknowledge} \\ &\land \texttt{SF\_SafeStop1\_1.Activate}) \\ &\implies EF\ \texttt{S\_Stopped}) \end{aligned} \tag{3}$$

Here, `Internal_Acknowledge` refers to an internal signal acknowledged by the motor once it has actually stopped.

Verifying the properties (1), (2), and (3) using the techniques described thus far is not possible, since the state space of the application is too large. We will, therefore, introduce additional abstractions that make use of the information inferred by the static analysis. In the end, these abstractions will allow us to verify the safety properties of this application.

9.1.1 *Modular Abstractions*

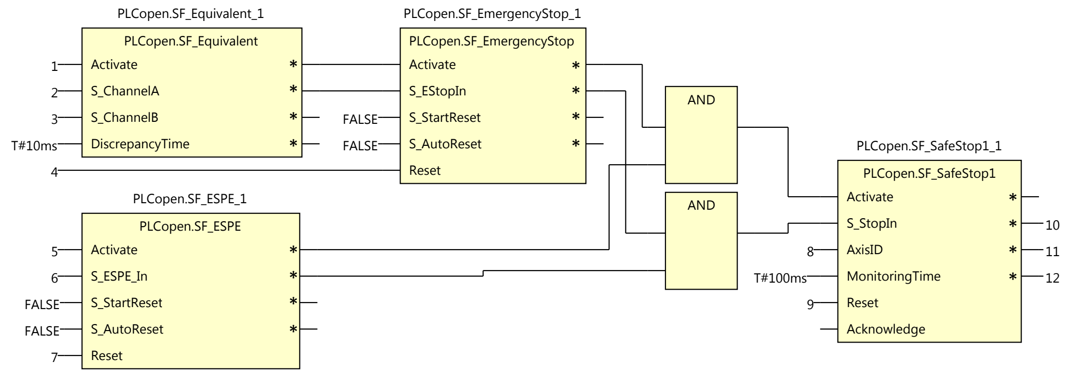
The structure of our properties (and, vice versa, the structure of the safety application) suggests that we modularly check only parts of the application. For property (1), e. g., evaluating the `SF_Equivalent` and `SF_EmergencyStop` blocks is strictly necessary. Due to the `AND` connecting to the other blocks, however, the application cannot easily be reduced using slicing techniques [121]. We will hence introduce a technique where we can selectively abstract blocks away or substitute them back in, based on the formula that is verified and potential counterexamples.

Modular Abstraction

The *modular abstraction* [13] replaces a call to a function block instance in a program by a summary of the effects of the call. This means that we first compute the summary of each function block using the static analysis, which over-approximates the potential ranges of the output variables. Instead of calling a function block in a program, we can now over-approximate the effect of the call by assigning the summary to the output variables (and other variables that are externally visible). We can thus save the execution of the FB and all refinements the execution would entail during verification. Technically, we introduce new input variables MA_i of the caller of the FB for each output of the called FB. These inputs are then only constrained to the summarized value-set of the called FB. On each call, the new MA_i variables are then copied in the variables of the abstracted FB. Consequently, the modular abstraction can only be applied to function block instances that are only called once per cycle (which is the typical usage).

Selecting Suitable Blocks

Of course, not all function blocks are suitable for such an abstraction. To find a good abstraction that is still precise enough for us to verify certain properties, we augment the counterexample-based refinement scheme introduced in Chap. 4: We first replace all function block calls. If then a counterexample depends on a summarized function block call, we replace it back and restart.



1	`InputDevice1_active`	2	`S1_S_EStopIn_1`
3	`S1_S_EStopIn_2`	4	`S0_RESET`
5	`InputDevice2_active`	6	`S2_S_ESPEIn`
7	`S0_RESET`	8	`AxisID_1`
9	`S3_DriveReset`	10	`S_Stopped`
11	`Error`		
12	`DiagCode`		

Figure 29: The abstracted example program.

This technique, however, is not sufficient to verify the safety application either. The reason is that we want to verify (1), we first get an abstracted program as shown in Fig. 29. This programs admits a spurious counterexample, but we cannot easily decide which block (`SF_EmergencyStop_1` or `SF_ESPE_1`) to replace first. Depending on whether we want to verify (1) or (2), the former or the latter is the correct choice. In the next section we describe a heuristic that allows us to automatically select the correct block for the refinement for both formulae.

9.1.2 *Selecting Modular Refinements using Forward Slicing*

Slicing [121, 119, 58] is a program analysis that extracts a part of program, a so-called *program slice*, w. r. t. a certain criterion. Such slicing criteria can be, e. g., a set of variables or a set of statements of the program. Using *forward slicing* one can then extract a slice that contains all parts of the program that can be influenced by the criterion. *Backward slicing*, on the other hand, will extract a slice of all parts of the program that can influence the property, i. e., the behavior of a certain criterion in the slice is indistinguishable from the original program. Slicing is performed by removing all program fragments that are—syntactically—independent from the behavior of the criterion in interest. To give a concrete example, assume a program is composed of two FBs that are not connected (neither via control nor via data flow). To verify a property about the input/output variables of one of the FBs, only that FB has to be considered since the FBs cannot influence each other. A slicer can detect this using backward slicing on the property in question.

For our analysis we use a slicer that works on the CFG of our IR, which was implemented independently of this work. It can perform forward and backward slicing for a set of program locations or for a set of program variables. When verifying properties, backward slicing is of special interest, for it returns a semantically equivalent program reduced to the property of interest. In particular when the safety function is part of a larger program, it allows to extract the relevant parts of the program, which will greatly reduce the state space.

The safety applications and functions block we analyzed in this work, however, were already reduced to their core safety function and not part of a larger application. All remaining functionality was heavily intertwined and hence, slicing did not have much effect in reducing the program size.

Forward Slicing

Instead of using backward slicing to produce a reduced program w. r. t. a safety property, we can also use forward slicing to analyze the influence of a property on the program. Here, we use forward slicing to select a block for refinement in the modular abstraction. Therefore, we compute the forward slice of all variables used in the specification. For formula (1), e. g., we would compute the forward slice of `S1_S_EStopIn_1`, `S1_S_EStopIn_2` and `SF_SafeStop1_1.S_StopIn`, which would indicate that these variables have influence on the `SF_Equivalent`, `SF_EmergencyStop` and `SF_SafeStop` block (as well as the `AND` block). The `SF_ESPE` block is not in the slice. Hence, we infer that we substitute the other blocks before we substitute the `SF_ESPE` block.

When proving formula (2), on the other hand, the `SF_Equivalent` and the `SF_-EmergencyStop` blocks are not in the slice, and hence the `SF_ESPE` block is substituted first. For this application, it turns out that this strategy is ideal in finding a suitable order of refinements. Yet, to verify the safety properties, we have to introduce another static analysis based abstraction.

9.1.3 *State Space Reduction using Liveness Analysis*

In Sect. 8.2.4 we introduced the LVA analysis, which determines the set of live variables for each program location. We used the results of this analysis to reduce the number of variables that have to be tracked during the static analysis.

This analysis is now used to further speed up the model checking process by reducing the state space. The key insight here is the following: Variables that are not live at the end of the cycle cannot influence the next program cycle, since their value is never read before being overwritten. We can, therefore, reset these variables to their default value in each program configuration. The effect of this is that states that differ only in dead variables only have to be handled once. This allows us to further abstract states reducing the size of the state space.

Since the value of dead variables is not read, this optimization is sound. Dead variables can, however, still play an important role if they are used in the specifi-

Property	#states	#transitions	Time
(1)	26 444	1 723 574	6 min 5 s
(2)	2 178	64 540	4 s
(3)	2 736	114 086	7 s

Table 9: Evaluation of the verification of the safety application.

cation. Therefore, we evaluate the specification before we reset the dead variables. After resetting these variables, states with an evaluation of the specification can be equal (since they only differ in dead variables). To distinguish between such states, we also store the evaluation of the atomic propositions as part of the PLC configuration. This incurs no further overhead since (a) the labeling of the atomic propositions is stored anyway as part of the model checking process, and (b) it will only discriminate between states when it is made necessary due to the specification.

Counterexample Replay

In practice, the state space reduction using liveness analysis will reset at least all input and temporary variables of the program. It will further reset other variables that are not explicitly marked as temporary. When a counterexample is found, the values of the input variables become especially crucial in understanding the counterexample. If their value is missing due to the abstraction, important information is missing. To make the counterexample more expressive, we have to use the counterexample analysis and replay techniques to add back the values of the dead variables as described in Sect. 4.5.1. As a result, the abstraction has no effect on the counterexamples when presented to the user.

9.1.4 *Final Analysis*

After enabling the modular abstraction with replacements selected by the forward slicing and the LVA-based state space reductions, we can now verify the safety application. When checking (1) with modular abstraction, we first abstract all FBs to their summary. Hence a counterexample is generated in the first step. Then, we select possible refinements using the forward slicer, which will substitute back the `SF_Equivalent` and `SF_EmergencyStop` blocks. The LVA based abstraction is now powerful enough to verify (1). Similar results are obtained for the other formulae.

The results proving all three properties for the safety application are shown in Tab. 9. We can now prove all safety properties in this application. The only manual step during the verification was the formalization of the properties in CTL. All abstractions steps were performed automatically using Arcade.PLC and the techniques described. It is important to observe that the conjunction of the properties cannot directly be verified, since the automatic abstraction cannot be performed as easily in this case. Therefore, we advocate to break down complex properties, such as the behavior of the safety application, into simplex sub-problems. These simplified formulae provide two advantages: They are more easily expressed in a formal language such as CTL or a safety automaton, and it is often easier to prove their validity using a model checker since simpler properties allow for more aggressive abstractions.

9.2 USING THE MODEL CHECKER TO AUGMENT STATIC ANALYSIS RESULTS

In this section we will now turn to enhancing the static analyzer results using the model checker. Since the static analyzer always works on an over-approximation of the program semantics, its results (e. g., in form of warnings) can be spurious. Additionally, it does not provide a proof, explanation or trace for its results. Often, this means that the user has to manually verify whether a warning is indeed legitimate or whether it originates from the over approximation. Even if it is legitimate, a trace to reproduce a failure is often desired.

In this case, the model checker can be used to produce such a trace, which explains and strengthens the results of the static analyzer. The results of the function block summary (cp. Sect. 8.7) can be analyzed in a straightforward way using the model checker: For each value (or for each range of values) the model checker can be asked to produce a witness for these values by checking, e. g., the formula $EF\ \texttt{var} = \mathit{value}$.

Detecting Unreachable Code

Finally, the model checker can also be used to prove the reachability of the statements of the program [112]. While the static analysis already provides a warning for unreachable lines, this warning will only catch unreachable lines in the over-approximated semantics. This means that a line marked unreachable by the static analysis is definitely unreachable, whereas a line that not marked might still not be reachable. Using the model checker, these properties can be verified for all lines such that all unreachable lines are caught. At the same time, the model checker provides a witness for each reachable line, which can aid test case generation [112, 30].

9.3 CONCLUSION

In this chapter, we connected the static analysis results and the model checker results to create more powerful analyses. First, we used the static analysis to summarize function blocks, which then allowed us to verify the safety function introduced in the first chapter. Crucial for this analysis was the modular abstraction. It works by abstracting function blocks by their summary, which are then only refined if they are influencing a specification. We determined this influence using counterexamples and forward slicing. Additionally, an LVA based abstraction that resets unused variables gave rise to a further reduction of the size of the state spaces. The key component of these techniques is to use the results of the static analysis in the model checker.

Secondly, we demonstrated how the model checker can be used to strengthen the results of the static analysis. It can, e. g., produce a witness for certain (illegal) values in variables or prove that the values are infeasible and only a result of the over-approximation of the static analysis. Additionally, it can be used to prove the reachability of the statements of the program. Ultimately, this can be integrated in a test case generation framework, where, e. g., a line coverage of the test cases is achieved using the model checker.

10

CONCLUSION

This dissertation studied the formal methods *model checking* and *static analysis* to check PLC programs for correctness. Therefore, we created the tool Arcade.PLC based on the existing [mc]square model checker. It provides both, automatic abstraction techniques to make industrial PLC programs or function blocks amenable for a formal analysis, but also a graphical user interface that guides the user in the application of formal methods. It thus tries to bridge the gap between the theory of formal methods and formal methods in practice.

10.1 FORMAL METHODS IN PRACTICE

In practice, the use of formal methods is hindered by technical limitations, i. e., memory or time constraints due to the complexity of the algorithm, but also usability issues, i. e., the manual effort to prepare and model the programs and formalize the specifications. In Arcade.PLC we therefore implemented features that automatically extract an abstracted model using CEGAR-techniques (Chap. 4) and a predicate abstraction (Chap. 5). Further, we implemented intuitive automata-based formalisms (Chap. 6) and automatic error localization techniques (Chap. 7) to make formal methods for PLC programs more accessible. Finally, we implemented a static analysis that detects common programming errors (Chap. 8) as a *push button* technology, which requires no manual effort from the user.

It is possible to use Arcade.PLC for the verification of function blocks, function block libraries, or programs. It can be used by vendors of function block libraries to check for problems using the static analysis, but also to verify that the function blocks conform to their specification using the model checker. Users of function block libraries can use the model checker to verify that the safety function is implemented correctly.

Another important aspect and design goal of Arcade.PLC is that it can be applied to incomplete source code, i. e., programs where some functions or variables are still missing. It is thus possible to apply our methods, especially the static analysis, early during the implementation phase. This gives feedback about potential problems or special cases that where not yet considered in the implementation process.

The formal methods implemented in Arcade.PLC hence achieve multiple goals, with increasing complexity for the user:

1. The static analysis is a light-weight technique that requires almost no effort from the user apart from importing the programs into Arcade.PLC. It can readily detect typical problems, for highly safety-critical as well as normal code. Inspecting the results of the static analysis still requires manual effort, but due to the low number of false positives we saw in practice, gives valuable feedback.

2. The model checker can be used to verify the behavior of function blocks. Using the automata-based specification formalisms introduced in Arcade.PLC, this requires only moderate effort from the user.

3. The model checker can be used to verify the safety function of an application using CTL. This step, however, usually requires deeper knowledge on how to specify the safety function and how to interpret the results.

To summarize, we think that the formal methods implemented in Arcade.PLC are ready to be deployed effectively in practice. Especially the static analysis can be used without additional effort while finding many problems early in the implementation phase.

10.2 FUTURE WORK

Currently, Arcade.PLC offers support for PLC programs written in Instruction List, Statement List, Structured Text, and Function Block Diagram. In the future, the missing languages Sequential Function Chart (SFC) and Ladder Diagram (LD) could be added. While a translation from SFC into our IR is possible and straightforward, the precision of the analysis of such a translation is likely to be very low. The reason for this is that SFC allows for multiple program locations to be active at the same time, which makes an analysis that works in a flow-sensitive way very imprecise: The analysis will infer that all program locations can be active and thus will propagate information between all steps of the SFC, independent of whether such transitions are actually possible. A static analysis geared towards analyzing SFCs, perhaps directly based on the representation of the SFC, could provide much better results since it could restrict the control flow to the set of possible transitions.

For FBDs, similar extensions might be an interesting line of research. Although FBDs are supported by the current approach, many warning of the static analysis are not directly transferable to FBDs. The warning for *unreachable code*, e. g., is not directly applicable to FBDs. Additionally, unreachable code is typical if the body of an FBD is implemented in ST but a Boolean input is hard-wired, which triggers the execution of this code. This is not necessary an error in the program, since the functionality of this block might just be unneeded for the application. Hence, we believe that more research is necessary to extract more suitable warnings for FBDs. Additional work in the user interface is required to present these warnings.

Further, a deeper integration into common PLC development tools might be a valuable extension in the future. Currently, exporting a PLC program from a development tool and then importing it back into ARCADE is a tedious task that takes valuable time. The benefits of an integration into a PLC development toolchain are threefold: First, it would simplify the general accessibility of the programs without the need for export and import functionality. At the same time, parsing of the programs could be easier by accessing internal data structures. Finally, it could provide a better (accustomed) user interface when presenting the problems and warnings.

From a technical standpoint, more advanced abstraction techniques could also be integrated into our framework. Recently, IC3-based[1] algorithms [38] were used for model checking invariants. These algorithm compute inductive invariants by repeated SAT or SMT solver calls. This line of research could be a valuable extension for the model checker of ARCADE.PLC.

A limitation of our current static analysis is that all abstract domains we implemented are non-relational. In the futures, relational domains such as convex polyhedra [46], difference bound matrices [80, 124], two variables per inequality [111] or octagons [87] could greatly enhance the precision of our results.

Finally, we recently developed a framework for automatic test case generation using our abstraction refinement in ARCADE.PLC [112, 30]. While the initial results look very promising, a systematic approach using concolic testing [110] techniques might generate better results.

1 IC3 stands for *Incremental Construction of Inductive Clauses for Indubitable Correctness*

BIBLIOGRAPHY

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Dissertation, DIKU, University of Copenhagen, Copenhagen, Denmark, 1994.

[2] F. Asteasuain and V. Braberman. Specification patterns can be formal and still easy. In *SEKE (International Conference on Software Engineering and Knowledge Engineering)*, pages 430–436, 2010.

[3] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14(3):293–340, 2007.

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[5] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2004.

[6] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 97–105, New York, NY, USA, 2003. ACM.

[7] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.

[8] M. Bani Younis and G. Frey. Formalization of existing PLC programs: A survey. In *CESA*, 2003.

[9] L. Baresi, M. Mauri, A. Monti, and M. Pezze. PLCTools: Design, formal validation, and code generation for programmable logic controllers. In *SMC*, pages 2437–2442, 2000.

[10] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of Satisfiability*, 185:825–885, 2009.

[11] E. Beckschulze, S. Biallas, and S. Kowalewski. Static analysis of lockless microcontroller C programs. In *Proceedings Seventh Conference on Systems Software Verification (SSV 2012)*, EPTCS, pages 103–114, 2012.

[12] D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 25–32. IEEE, 2009.

[13] S. Biallas, D. Bohlender, and S. Kowalewski. Boolean and modular abstractions for programmable logic controllers. In *Dependable Control of Discrete Systems (DCDS'13)*, pages 97–102. IEEE, 2013.

[14] S. Biallas, J. Brauer, D. Gückel, and S. Kowalewski. On-the-fly path reduction. *Electronic Notes in Theoretical Computer Science*, 274C:3–16, 2011. 4th International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2010).

[15] S. Biallas, J. Brauer, A. King, and S. Kowalewski. Loop leaping with closures. In Antoine Miné and David Schmidt, editors, *19th Static Analysis Symposium*, Lecture Notes in Computer Science, pages 214–230. Springer Berlin Heidelberg, 2012.

[16] S. Biallas, J. Brauer, and S. Kowalewski. Counterexample-guided abstraction refinement for PLCs. In *5th International Workshop on Systems Software Verification (SSV 2010), Vancouver, Canada*, pages 2–12, Berkeley, CA, USA, 2010. USENIX Association.

[17] S. Biallas, J. Brauer, and S. Kowalewski. Sat-based abstraction refinement for programmable logic controllers. In *Dependable Control of Discrete Systems (DCDS'11)*, pages 96–101. IEEE, 2011.

[18] S. Biallas, J. Brauer, and S. Kowalewski. Arcade.PLC: A verification platform for programmable logic controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 338–341. ACM, 2012.

[19] S. Biallas, J. Brauer, S. Kowalewski, and B. Schlich. Automatically deriving symbolic invariants for PLC programs written in IL. In Eckehard Schnieder and Geza Tarnai, editors, *FORMS/FORMAT 2010*, pages 237–245. Springer Berlin Heidelberg, 2011.

[20] S. Biallas, G. Frey, S. Kowalewski, B. Schlich, and D. Soliman. Formale Verifikation von Sicherheits-Funktionsbausteinen der PLCopen auf Modell- und Code-Ebene. In *Tagungsband Entwicklung und Betrieb komplexer Automatisierungssysteme (EKA 2010)*, pages 49–57. ifak Magdeburg, 2010.

[21] S. Biallas, N. Friedrich, H. Simon, and S. Kowalewski. Automatic error cause localization of faulty PLC programs. In *Dependable Control of Discrete Systems (DCDS'15)*, volume 48, pages 79–84. Elsevier Ltd, 2015.

[22] S. Biallas, M. Giacobbe, and S. Kowalewski. Predicate abstraction for programmable logic controllers. In *18th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2013)*, pages 123–138, 2013.

[23] S. Biallas, V. Kamin, S. Kowalewski, B. Schlich, S. Sehestedt, and S. Stattelmann. Verifikation von sicherheitsgerichteten SPS-Programmen mit Hilfe von Safety-Automaten. In VDI Wissensforum, editor, *Automation 2013*, VDI Berichte. VDI-Verlag, 2013.

[24] S. Biallas, S. Kowalewski, and B. Schlich. Leistungsfähige Verifikation von industriellen SPS-Programmen mittels Model-Checking und statischer Analyse. In *AUTOMATION 2011, Baden-Baden, Germany*, number 2143 in VDI-Berichte, pages 67–72, Düsseldorf, 2011. VDI-Verlag.

[25] S. Biallas, S. Kowalewski, and B. Schlich. Automatische Wertebereichsanalyse – Formale Verifikation für SPS-Programme. *Automatisierungstechnische Praxis (atp EDITION), 54. Jahrgang, 7-8/2012*, pages 68–74, 2012.

[26] S. Biallas, S. Kowalewski, and B. Schlich. Automatische Wertebereichsanalyse von SPS-Programmen. In *AUTOMATION 2012, Baden-Baden, Germany*, number 2171 in VDI-Berichte, pages 79–83, Düsseldorf, 2012. VDI-Verlag. Long version (12 pages) on CD-ROM.

[27] S. Biallas, S. Kowalewski, and B. Schlich. Range and value-set analysis for programmable logic controllers. In *Proceedings of the 11th International Workshop on Discrete Event Systems*, pages 378–383, Guadalajara, Mexico, 2012. IFAC.

[28] S. Biallas, S. Kowalewski, S. Stattelmann, and B. Schlich. Efficient handling of states in abstract interpretation of industrial programmable logic controller code. In *Proceedings of the 12th International Workshop on Discrete Event Systems*, pages 400–405, Cachan, France, 2014. IFAC.

[29] S. Biallas, M. Chr. Olesen, F. Cassez, and R. Huuck. Ptrtracker: Pragmatic pointer analysis. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013)*, 2013.

[30] S. Biallas, H. Simon, S. Kowalewski, S. Hauck-Stattelmann, and B. Schlich. Automatische testfallgenerierung für sps-programme mittels zeilenüberdeckung. In *AUTOMATION 2015*, pages 100–111. VDI, 2015.

[31] F. Bitsch. *Verfahren zur Spezifikation funktionaler Sicherheitsanforderungen für Automatisierungssysteme in Temporallogik*. PhD thesis, Universität Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2007.

[32] D. Bohlender. Bachelor thesis: Modulare und Boolesche Abstraktion von SPS-Programmen, 2013. Lehrstuhl für Informatik 11, RWTH Aachen University.

[33] S. Bornot, R. Huuck, B. Lukoschus, and Y. Lakhnech. Utilizing static analysis for programmable logic controllers. In *ADPM*, pages 183–187, 2000.

[34] A. Braining. Master thesis: Model-Checking Automaten-basierter Spezifikationen für eingebettete Systeme, 2013. Lehrstuhl für Informatik 11, RWTH Aachen University.

[35] J. Brauer, T. Noll, and B. Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *SCOPES 2010*. ACM, 2010.

[36] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of PLC programs written in instruction list. In *2000 IEEE International Conference on Systems, Man, and Cybernetics, Nashville, TN, USA*, volume 4, pages 2449–2454. IEEE Computer Society Press, 2000.

[37] L. Chen and W. L. Harrison III. An efficient approach to computing fixpoints for complex program analysis. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, pages 98–106. ACM, 1994.

[38] A. Cimatti and A. Griggio. Software model checking via IC3. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 277–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[39] E. M. Clarke. *The Birth of Model Checking*. Springer-Verlag, Berlin, Heidelberg, 2008.

[40] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV 2000), Chicago, USA*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[41] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001.

[42] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[43] A. Cortesi. Widening operators for abstract interpretation. In *Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40, 2008.

[44] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[45] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6:69–95, 1999.

[46] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[47] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, pages 451–590, 1991.

[48] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[49] D. Darvas, B. Adiego, A. Vörös, T. Bartha, E. Viñuela, and V. Suárez. Formal verification of complex properties on PLC programs. In *34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems (FORTE)*, pages 284–299, 2014.

[50] H. Dierks. PLC-Automata: A New Class of Implementable Real-time Automata. *Theor. Comput. Sci.*, 253(1):61–93, February 2001.

[51] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3:131–165, 1994.

[52] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[53] E. A. Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter Temporal and Modal Logics, pages 995–1072. The MIT Press, 1991.

[54] E. Ermis, M. Schäf, and T. Wies. Error invariants. In *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 187–201. Springer Berlin Heidelberg, 2012.

[55] G. Frey, B. Schlich, R. Drath, and R. Eschbach. Safety automata – A new specification language for the development of PLC safety applications. In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–8, Sept 2012.

[56] N. Friedrich. Bachelor thesis: Precise Counterexamplegeneration for Programmable Logic Controllers, 2013. Lehrstuhl für Informatik 11, RWTH Aachen University.

[57] V. Gafni. About the Compilation of CSL, a Real-Time — pattern based — Specification Language. `http://www.cs.tau.ac.il/~amiramy/SoftwareSeminar/CSL_TAU_Talk_July_09.ppt`. Accessed: 2015-09-14.

[58] K. Gallagher and D. Binkley. Program slicing. In *Frontiers of Software Maintenance*, pages 58–67, Sept 2008.

[59] M. Giacobbe. Master thesis: Predicate Abstraction of PLC Programs using SMT Solving, 2013. Lehrstuhl für Informatik 11, RWTH Aachen University.

[60] T. Goldschmidt, M. Murugaiah, C. Sonntag, B. Schlich, S. Biallas, and P. Weber. Cloud-based control: A multi-tenant, horizontally scalable soft-PLC. In *CLOUD 2015*, 2015.

[61] T. Goldschmidt, M. Murugaiah, C. Sonntag, B. Schlich, S. Biallas, and P. Weber. Cloud-basierte Steuerungen: Eine horizontal skalierbare, multi-tenant-fähige Soft-SPS. In VDI Wissensforum, editor, *Automation 2015*, VDI Berichte. VDI-Verlag, 2015.

[62] V. Gourcuff, O. De Smet, and J. M. Faure. Efficient representation for formal verification of PLC programs. In *8th International Workshop on Discrete Event Systems*, pages 182–187, 2006.

[63] V. Gourcuff, O. De Smet, and J.-M. Faure. Improving large-sized PLC programs verification using abstractions. In *Proceedings of the 17th IFAC World Congress*, pages 5101–5106, 2008.

[64] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[65] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, June 2006.

[66] D. Gückel. *Synthesis of State Space Generators for Model Checking Microcontroller Code*. Dissertation, Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen, November 2014.

[67] D. Gückel and S. Kowalewski. Automatic derivation of abstract semantics from instruction set descriptions. In *Proceedings of the 6th International Workshop on Systems Software Verification (SSV 2011)*, pages 18–32. TU Dresden, 2011.

[68] S. Hauck-Stattelmann, S. Biallas, B. Schlich, S. Kowalewski, and R. Jetley. Analyzing the restart behavior of industrial control applications. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 585–588. Springer International Publishing, 2015.

[69] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[70] International Electrotechnical Commission. *IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems*. International Electrotechnical Commission, Geneva, Switzerland, 1998.

[71] International Electrotechnical Commission. *IEC 61131: Programmable Controllers*. International Electrotechnical Commission, Geneva, Switzerland, 2003.

[72] International Electrotechnical Commission. *IEC 60848: GRAFCET specification language for sequential function charts*. International Electrotechnical Commission, Geneva, Switzerland, 2013.

[73] International Electrotechnical Commission. *IEC 62714: Engineering data exchange format for use in industrial automation systems engineering - Part 1: Architecture and General Requirements*. International Electrotechnical Commission, Geneva, Switzerland, 2014.

[74] S. Konrad and B.H.C. Cheng. Facilitating the construction of specification pattern-based properties. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 329–338, Aug 2005.

[75] B. Kormann and B. Vogel-Heuser. Automated test case generation approach for PLC control software exception handling using fault injection. In *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pages 365–372, 2011.

[76] T. Kumazawa and T. Tamai. Counterexample-based error localization of behavior models. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 222–236. Springer Berlin Heidelberg, 2011.

[77] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA, 1994.

[78] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.

[79] F. Laroussinie, A. Meyer, and E. Petonnet. Counting CTL. In Luke Ong, editor, *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 206–220. Springer Berlin Heidelberg, 2010.

[80] K.G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24, Dec 1997.

[81] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[82] O. Ljungkrantz, K. Åkesson, Chengyin Yuan, and M. Fabian. Towards industrial formal specification of programmable safety systems. *Control Systems Technology, IEEE Transactions on*, 20(6):1567–1574, Nov 2012.

[83] K. L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[84] T. Mertke. *Formale Spezifikation reaktiver Systeme mit einer Sicherheitsfachsprache*. Dissertation, Brandenburgisch Technische Universität Cottbus, August 2004.

[85] T. Mertke and G. Frey. Formal verification of PLC-programs generated from signal interpreted petri nets. In *2001 IEEE International Conference on Systems, Man, and Cybernetics, Tuscon, AZ, USA*, volume 4, pages 2700–2705. IEEE Computer Society Press, 2001.

[86] T. Mertke and T. Menzel. Methods and tools to the verification safety-related control software. In *SMC*, pages 2455–2457, 2000.

[87] A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.

[88] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2):53–59, 1994.

[89] L. Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[90] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[91] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[92] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *Hardware and Software: Verification and Testing (HVC 2007), Haifa, Israel*, volume 4899 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2008.

[93] O. Pavlovic, R. Pinger, and M. Kollmann. Automated formal verification of PLC programms written in IL. In *4th International Verification Workshop (VERIFY'07), Bremen, Germany*, number 259 in CEUR Workshop Proceedings, pages 152–163. CEUR-WS.org, 2007.

[94] PLCopen TC5. *Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks*. PLCopen, Germany, 2006.

[95] H. Prähofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger. Opportunities and challenges of static code analysis of IEC 61131-3 programs. In *IEEE 17th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, Sept 2012.

[96] R. Ramler, W. Putschögl, and D. Winkler. Automated testing of industrial automation software: Practical receipts and lessons learned. In *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, MoSEMInA 2014, pages 7–16, New York, NY, USA, 2014. ACM.

[97] J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *ACM SIGPLAN/SIGBED Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES 2006), Ottawa, Canada*, pages 34–43. ACM, 2006.

[98] J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.

[99] T. Reinbacher and J. Brauer. Precise control flow reconstruction using boolean logic. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and S. Fischmeister, editors, *International Conference on Embedded Software (EMSOFT 2011)*, pages 117–126. ACM, 2011.

[100] T. Reinbacher, J. Brauer, M. Horauer, and B. Schlich. Refining assembly code static analysis for the Intel MCS-51 microcontroller. In *Industrial Embedded Systems (SIES'09), Lausanne, Switzerland*, pages 161–170. IEEE Computer Society Press, 2009.

[101] T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer. Model checking embedded software of an industrial knitting machine. *International Journal of Information Technology, Communications and Convergenceonvergence*, pages 186–205, 2010.

[102] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pages 30–39, 2003.

[103] B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

[104] B. Schlich, J. Brauer, and S. Kowalewski. Application of static analyses for state space reduction to microcontroller binary code. *Sci. Comput. Program.*, 76(2):100–118, 2011.

[105] B. Schlich, J. Brauer, J. Wernerus, and S. Kowalewski. Direct model checking of PLC programs in IL. In *Dependable Control of Discrete Systems (DCDS'09), Bari, Italy*, pages 28–33, 2009.

[106] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2009.

[107] T. Schlipf, T. Buechner, R. Fritz, M. M. Helms, and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 41(4&5):567–576, 1997.

[108] A. Schumacher. Bachelor thesis: Verifikation von STL-Programmen mit [mc]square, 2011. Lehrstuhl für Informatik 11, RWTH Aachen University.

[109] H. Seidl, R. Wilhelm, and S. Hack. *Compiler Design – Analysis and Transformation*. Springer, 2012.

[110] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE 05)*, pages 263–272. ACM Press, 2005.

[111] A. Simon and A. King. The two variable per inequality abstract domain. *Higher-Order and Symbolic Computation*, 23(1):87–143, 2010.

[112] H. Simon, N. Friedrich, S. Biallas, S. Hauck-Stattelmann, B. Schlich, and S. Kowalewski. Automatic test case generation for PLC programs using coverage metrics. In *ETFA*, 2015. To appear.

[113] A. Smith, A. Veneris, M. F. Ali, and A. Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 24(10):1606–1621, November 2006.

[114] D. Soliman and G. Frey. Verification and Validation of Safety Applications based on PLCopen Safety Function Blocks using Timed Automata in Uppaal. In *2nd IFAC Workshop on Dependable Control of Discrete Systems (DCDS)*, 2009.

[115] S. Stattelmann, S. Biallas, B. Schlich, and S. Kowalewski. Applying static code analysis on industrial controller code. In *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2014. To appear.

[116] A. Sülflow and R. Drechsler. Verification of PLC programs using formal proof techniques. In G. Tarnai and E. Schnieder, editors, *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/-FORMAT 2008), Budapest, Hungary*, pages 43–50, Budapest, Hungary, 2008. L'Harmattan.

[117] A. Sülflow and R. Drechsler. Automatic fault localization for programmable logic controllers. In Eckehard Schnieder and Géza Tarnai, editors, *FORMS/-FORMAT*, pages 247–256. Springer, 2010.

[118] R. Šusta. *Verification of PLC Programs*. PhD thesis, CTU-FEE Prague, 2002.

[119] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[120] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

[121] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE 81), San Diego, USA*, pages 439–449. IEEE Press, 1981.

[122] B. Wichmann, A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, pages 69–75, 1995.

[123] W. E. Wong and V. Debroy. A survey of software fault localization, 2009. Technical Report UTDCS-45-09. Department of Computer Science. The University of Texas at Dallas.

[124] S. Yovine. Model checking timed automata. In *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer Berlin Heidelberg, 1998.

AACHENER INFORMATIK-BERICHTE

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2013-01 *	Fachgruppe Informatik: Annual Report 2013
2013-02	Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen
2013-03	Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM
2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
2013-05	Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013
2013-06	Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation
2013-07	André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung
2013-08	Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers
2013-10	Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata
2013-12	Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs
2013-13	Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-14	Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures
2013-16	Carsten Otto: Java Program Analysis by Symbolic Execution
2013-19	Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol
2013-20	Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models
2014-01 *	Fachgruppe Informatik: Annual Report 2014
2014-02	Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03	Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04	Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06	Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08	Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09	Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14	Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15	Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16	Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 *	Fachgruppe Informatik: Annual Report 2015
2015-02	Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05	Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06	Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07	Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08	Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09	Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models
2015-11	Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus
2015-12	Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic
2015-13	Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2015-14	Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines
2016-01 *	Fachgruppe Informatik: Annual Report 2016
2016-02	Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems
2016-03	Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04	Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05	Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization
2016-06	Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.