Predicate Abstraction for Programmable Logic Controllers

Sebastian Biallas, Mirco Giacobbe and Stefan Kowalewski

Embedded Software Laboratory, RWTH Aachen University, Germany

Abstract. In this paper, we present a predicate abstraction for programs for programmable logic controllers (PLCs) so as to allow for model checking safety related properties. Our contribution is twofold: First, we give a formalization of PLC programs in first order logic, which is then used to automatically derive a predicate abstraction using SMT solving. Second, we employ an abstraction called predicate scoping which reduces the evaluation of predicates to certain program locations and thus can be used to exploit the cyclic scanning mode of PLC programs. We show the effectiveness of this approach in a small case study using programs from industry and academia.

1 Introduction

Programmable Logic Controllers (PLCs) [13] are control devices used in industry to monitor, operate and control various machines, robots, assembly lines, chemical plants, power plants and oil rigs. PLCs comprise input connectors (typically connected to sensors), output connectors (typically connected to actuators) and a program which controls the operation. In the most common mode of operation, the so-called *cyclic scanning mode*, three phases are executed atomically: (a) the current values at the input connectors are stored in input variables, (b) the program is called to determine new values for the output variables and (c) the values of the output variables are copied to the output connectors. By executing these phases repeatedly at a high frequency the connected machinery can be controlled in a closed loop. Additionally to the input and output variables, the program can access non-temporal variables whose values are retained for the next cycle. The program itself is usually composed of functional components called *function blocks* (FBs), which can be written in various languages [13, Part 3].

Since PLCs operate in highly critical environments, it is recommended to verify their programs using formal methods [14]. In this paper we advocate model checking to verify functional properties.

1.1 Model Checking PLC Programs

Typical properties to be checked relate input values to output values. One might want to check, e. g., an invariant such as: If an emergency stop is signaled (input is set), the motor is stopped (output is not set). Other formulae might refer to

```
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
```

Fig. 1. Example PLC program used throughout the paper

the order of certain events, e. g., after a failure, the motor does not start until the failure is acknowledged. For such properties ∀CTL is a suitable formalism. This approach is already implemented in the ARCADE.PLC framework [5] on which our work builds. Yet, the implementation is limited due to the availability of non-relational domains only and the restrictions of a hand written constraint solver, which then results in a state explosion for more complicated programs.

1.2 Contribution & Outline

To overcome these obstacles we propose a fully automatic predicate abstraction. In the underlying principles of this approach, we first encode the semantics of a given PLC program as a first order logic formula (Sect. 3). In Sect. 4, we then automatically derive the transition relation of the abstracted state space and introduce the predicate scoping to further reduce state spaces. The feasibility of our approach is demonstrated in Sect. 5 on various PLC programs. The paper ends with a discussion of related work in Sect. 6 and a conclusion in Sect. 7. We start by introducing our worked example.

2 Worked Example

We motivate our approach with the small example program shown in Fig. 1. This program is written in *structured text*, one of the five standard PLC programming languages [13]. Each cycle, i. e., each time the program is called, the following operation is performed: The input `flag` is tested (line 12). If the flag is not set then the output variable `out` is set to the value of `var` (line 19). Otherwise, the program tests whether the sum of the three inputs `in0`, `in1`, `in2` is less than 100 (line 13). If so, `in0` is assigned to `var`, otherwise `var` is set to 0. Note that `var` is a non-temporary variable, which retains its value for the next cycle.

Suppose we want to verify the program invariant `out` < 100. Careful inspection of the program reveals that this invariant is true: the variable `out` is only set to

the value of `var`, which in turn is either set to 0 or `in0`. The first case is trivial, the second case is only executed if $\texttt{in0} + \texttt{in1} + \texttt{in2} < 100$ which implies that $\texttt{in0} < 100$ (overflow can not occur here, since arithmetic is implicitly cast to a bigger accumulator data type here). Note that it is not obvious how to prove this invariant automatically.

Our approach works by model checking the state space of a PLC program, which allows to not only check simple invariants but also the correct ordering of certain events. The state space itself comprises of states that are tuples of the current line number and the variable values (we will defer a formal introduction to Sect. 3). It is important to observe that during the execution of the program the PLC does not communicate with input/output connectors. All communication is performed atomically at the beginning/end of the cycle so as to allow preliminary values during the computation. If, e. g., the program sequentially sets a number of outputs, all intermediate states are not observable until the end of the program and thus should not be subject to the verification. Therefore, all properties should only be checked in the very last program line. The invariant we want to verify will hence be encoded in ∀CTL as

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

where exitpoint evaluates to true only at exit. Such a property could naïvely be checked by enumerating the complete concrete state space. Yet, even for the small example program, this state space would have an excess amount of states making this approach unfeasible in practice. We therefore turn to a predicate abstraction where sets of concrete states are abstractly represented by a predicate (or conjunctions of predicates).

3 Encoding of PLC semantics in FOL

In this section, we describe the transformation of PLC programs into first order logic (FOL) formulae. These formulae are written in a way that each model of these formulae represents a possible state change by statement encoded in the formula. Syntactically, this is indicated by using unprimed variables as pre- and primed variables as post-variables for encoding the semantics. This allows to derive the transition relation between (possible abstract) states with decision procedures. Later, we will then combine formulae describing a single step of the program into formulae describing the application of a basic block or even a whole cycle. We start by encoding the different types of variables.

3.1 Encoding of Variables and the Program

Let Var be the set of variables of the PLC program after flattening all structures and arrays. Depending on their lifetime and their semantics, we partition Var into three distinct sets:

– Var_M contains variables that retain their value between cycles. Such variables are defined, e. g., using the keywords `VAR` or `VAR_GLOBAL`.

– The set $\textsc{Var}_I$ holds the input variables. These are the variables declared as `VAR_INPUT` or `VAR_INOUT` in the outmost function block or in the program (input variables of inner function block instances are determined at the call site and thus not non-deterministic).
– Finally, all temporary variables are contained in $\textsc{Var}_T$. These variables are reinitialized to their default value each cycle and thus do not retain their value for the next cycle.

Note that recursion is not possible in PLC programs. We can thus determine the number of variables used in total and do not need special handling of local variables.

Definition 1 (Memory). *The PLC memory state is defined as a tuple* $\langle \textsc{Var}, \mathcal{D} \rangle$ *where* $\mathcal{D}$ *is the domain of discourse.*

The domain of discourse can be selected as the union of the data types of the variables in Var, which after flattening are all elementary (i. e., non-aggregate) data types.

Definition 2 (Memory State). *A program 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 *contains a symbolic address (e. g., a line number) of the next statement to be executed.*

We define two special program locations ℓ_a and ℓ_z which denote the first and last statement, respectively (the program can always be transformed to have only one exit point). A memory state uniquely determines which operation(s) are performed next and their result. We thus can define all possible executions of the PLC program in the following form:

Definition 3 (Program Model). *A 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.*

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

Since the state of a program is given by an assignment to all variables, the number of possible states explodes exponentially in the number of variables. A symbolic encoding of transition systems allows us to represent huge — potentially infinite — sets of states by describing them using some language. In this work, we use the first-order logics to represent sets of states, where each variable represents a declared variable of the program.

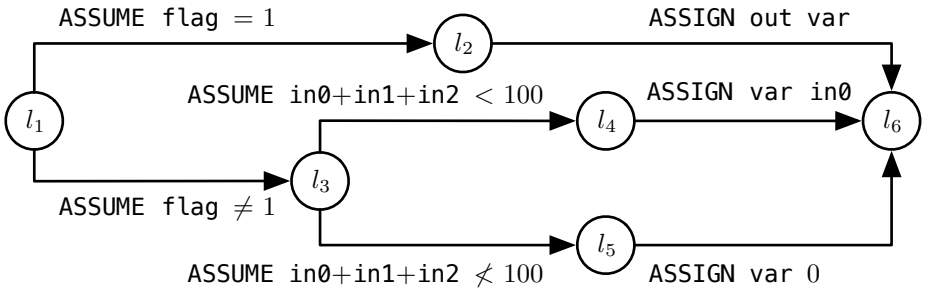

Fig. 2. Control-flow automaton of program `Example`.

3.2 From PLC Programs to FOL Formulae

Our transformation is performed in two steps. First, we translate the program statements into a control-flow automaton [3] which itself uses an intermediate representation of the program statements. Then, we will transform the automaton into formulae.

Definition 4 (Control-flow automaton). *A control-flow automaton (CFA) is a labeled state transition system* $\langle L, \text{Ops}, G \rangle$ *where* L *is a set of program locations,* Ops *is a set of operations over the memory and* $G \subseteq L \times \text{Ops} \times L$ *is a set of control-flow edges.*

Each $\ell \in L$ represents a location in the program. The semantics of a control-flow edge $\langle \ell, \cdot, \ell' \rangle$ is that if the current state is in location ℓ, then the next state will be in location ℓ' after the execution of the operation. An operation is given as one of the following instructions:

- `ASSUME` π
- `ASSIGN` $x\ t$

The operation *assume* is followed by a predicate application π with positive $p(t_1, \ldots, t_n)$ or negative $\bar{p}(t_1, \ldots, t_n)$ polarity and it means that the transition is executed if and only if the given predicate application is valid or not valid, respectively. The operation *assign* is followed by a variable x and a term t of the same type and it means that in the next state the variable x will be assigned with the evaluation of the expression t in the current state. A term t can be a variable or a function application:

Definition 5 (Term). *The rules for forming a term* t *are defined by the following abstract grammar:*

$$t ::= \quad x \quad | \quad f(t_1, \ldots, t_n)$$

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

Note that constants are applications of functions with arity 0. We allow all standard PLC operators as functions, which also includes casts to different scalar data types.

Example 2. Consider the program `Example` of Fig. 1 and the memory defined as in Ex. 1. The labeled state transition system of Fig. 2 represents the translation into a control-flow automaton as in Def. 4. For simplicity 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 $<(+(+(\texttt{in0}, \texttt{in1}), \texttt{in2}), 100)$.

In order to be manipulated by SMT solvers, memory operations must be translated into quantifier-free first-order logic formulae that encode in some way assumptions and transformations over the memory. Quantifier-free first-order logics formulae can be expressed in terms of predicate applications, negations and conjunctions.

Definition 6 (Quantifier-free Formula). *The formation rules of a formula φ are defined by the following abstract grammar:*

$$\varphi ::= \quad p(t_1, \ldots, t_n) \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2$$

where p is a predicate symbol and $t_1, \ldots, t_n$ are terms (cp. Def. 5).

Operations are encoded as cubes over equalities between *post-variables* and terms over *pre-variables*, representing valuations of pre- and post-states of the transition. Pre- and post-variables are denoted by the set of unprimed and primed symbols respectively. Let $\mathcal{L}$ be the language of first-order logic. We define an operation encoder as a function $\epsilon\colon \textsc{Ops} \to \mathcal{L}$. According to its semantics, 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:

$$\begin{aligned} \epsilon(\texttt{ASSUME}\ p(t_1, \ldots, t_n)) &\triangleq p(t_1, \ldots, t_n) \wedge \textstyle\bigwedge_{y \in \textsc{Var}} y' = y \ , \\ \epsilon(\texttt{ASSUME}\ \bar{p}(t_1, \ldots, t_n)) &\triangleq \neg p(t_1, \ldots, t_n) \wedge \textstyle\bigwedge_{y \in \textsc{Var}} y' = y \ . \end{aligned}$$

An assignment is encoded by asserting the equality between the post-variable that has to be assigned with the term over the pre-variables, conjuncted with equalities between pre- and post- versions of the variables that have to remain unchanged, which are all except for the assigned one:

$$\epsilon(\texttt{ASSIGN}\ x\ t) \triangleq x' = t \wedge \textstyle\bigwedge_{y \in \textsc{Var} \setminus \{x\}} y' = y \ .$$

A first-order logic control-flow-based symbolic encoding of a PLC program is then given by the following:

- $\langle \textsc{Var}, \mathcal{D} \rangle$ the memory and $\textsc{Var}_I$, $\textsc{Var}_M$, $\textsc{Var}_T$ the variable classes,

– $\langle L, \mathcal{L}, G \rangle$ the CFA where all operations in Ops are encoded into the first-order logic language $\mathcal{L}$ through the encoder ϵ,
– $\mathit{Start} \in L \times \mathcal{L}$ the start-up phase and
– $\mathit{Scan} \in L \times \mathcal{L}$ the scanning phase.

The *start-up phase* defines the initialization in the initial location ℓ_a:

$$\mathit{Start} \triangleq \langle \ell_a, \bigwedge_{x \in \textsc{Var} \setminus \textsc{Var}_I} x = \mathsf{Init}_x() \rangle$$

where $\mathsf{Init}_x()$ is the constant x is initialized to. The *scanning phase* defines the behavior of the controller after the execution of the program body after it reaches the last program location $\ell_z \in L$. 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 is non-deterministic:

$$\mathit{Scan} \triangleq \langle l_z, \bigwedge_{x \in \textsc{Var} \setminus \textsc{Var}_I} x' = x \rangle$$

We obtained a symbolic encoding of a program $\langle S, I, R \rangle$ as in Def. 3 that can be handled with SMT solving techniques [2]. Given a theory $\mathcal{T}$ chosen for the interpretation of variables and predicates, the set of states is defined as the set of all locations and all possible assignments consistent under $\mathcal{T}$:

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

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

$$I \triangleq \{\langle \ell_a, \nu \rangle \mid \nu \models_{\mathcal{T}} \varphi \text{ and } \mathit{Start} = \langle \ell_a, \varphi \rangle\}$$

The transition relation is given by pairs of states over 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' together satisfy the formula. Moreover, all transitions are those pairs of states from the last to the first location that satisfy the scanning phase:

$$\begin{aligned} R \triangleq\ & \{\langle \langle \ell, \nu_1 \rangle, \langle \ell', \nu_2 \rangle \rangle \mid \nu_1, \nu_2' \models_{\mathcal{T}} \varphi \text{ and } \langle \ell, \varphi, \ell' \rangle \in G\} \ \cup \\ & \{\langle \langle \ell_z, \nu_1 \rangle, \langle \ell_a, \nu_2 \rangle \rangle \mid \nu_1, \nu_2' \models_{\mathcal{T}} \varphi \text{ and } \mathit{Scan} = \langle \ell_z, \varphi \rangle \text{ and } \mathit{Start} = \langle \ell_a, \cdot \rangle\}. \end{aligned}$$

3.3 Encoding of Timers

PLC programs are allowed to use the timer FBs `TP`, `TON` and `TOF`, which by definition have continuous behavior, which cannot be encoded by the default operation set. For this reason, we augment Ops with these operations to represent timer calls:

– $\mathtt{TP}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}$, – $\mathtt{TON}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}$, – $\mathtt{TOF}\ n\ t_{\mathtt{IN}}\ t_{\mathtt{PT}}$,

where $n \in$ Timer is the name of the timer and $t_{\mathtt{IN}}$ (timer input) and $t_{\mathtt{PT}}$ (programmed time) are terms. For each timer $n \in$ Timer the variables $n.\mathtt{IN}$ (timer input), $n.\mathtt{Q}$ (timer output) are added to Var. For timers of type TON and TOF the propositional variable $n.r$ is added to Var, which keeps track whether the timer is running. None of those variables is added to any variables class.

The rising and falling edges on the input and the relative instant behavior are encoded at the call site. Timer TP starts (setting Q to 1) if Q is 0 and there is a rising edge on input IN; nothing is changed otherwise:

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

Timer TON starts on rising edges of IN and stops (setting Q to 0) on falling edges:

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

Timer TOF starts on falling edges and stops setting Q to 1 on rising edges and is defined similar to TON. At the start-up phase and the scanning phase are augmented to encoded initialization and time elapsing, respectively. Timers are initially disabled and with all values set to 0. Timer TP *may* have a falling edge on Q if running, it remains disabled with Q set to 0 otherwise:

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

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

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

3.4 Summarization of Control-Flow Automata

The obtained control-flow automaton contains one transition for every instruction of the intermediate language. This kind of encoding is called *single-block encoding* (SBE) and it has two drawbacks: First, one has to compute the evaluation of all formulae at each intermediate step, even if these evaluations bear no further meaning. Secondly, the conjunction of intermediate steps (formulae) might be easier to evaluate using automated decision procedures than each step on its

own—it might even be more precise. Such a conjunction of simple intermediate steps is called *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) [3]. We use a BBE in our approach.

4 Predicate Abstraction

Let $P = \{\pi_1, \ldots, \pi_n\}$ a set of predicates over the set of concrete variables VAR, which we call the *abstraction precision*. The Boolean predicate abstraction of a system constitutes computing an over-approximation which keeps track where each of the predicates in P is valid or not. An abstract state is defined as a pair $\langle \ell, c \rangle$ of location ℓ and a minterm c over a set $B = \{b_1, \ldots, b_n\}$ of only propositional variables. A minterm over B is a conjunction of all variables $b_i \in B$, each of them occurring either with positive b_i or with negative $\neg b_i$ polarity. We define the *abstraction function* α of a concrete state $\langle \ell, \nu \rangle$ as the abstract state $\langle \ell, c \rangle$ in which the polarity of each variable in c states the validity of the respective predicate in ν:

$$\alpha(\langle \ell, \nu \rangle) \triangleq \langle \ell, c \rangle \text{ s.t. } c \models b_i \text{ iff } \nu \models_{\mathcal{T}} \pi_i \text{ for each } 1 \leq i \leq n$$

The abstraction function creates an over-approximation, meaning that the abstraction of a state s represents a region of states in which s is contained, i. e., $s \in (\alpha \circ \alpha^{-1})(s)$. Note that α is a surjection, hence we are abusing the notation defining its inverse as $\alpha^{-1}(\hat{s}) \triangleq \{s \mid \hat{s} = \alpha(s)\}$.

We are interested in verifying universal properties and we want the abstraction to be conservative for such properties. This is guaranteed by assuring all predicates of the property to be contained in the precision. In this way, the abstraction of the region of states S_ϕ in which the property ϕ is satisfied is represented with the highest precision possible, i. e., $S_\phi = \alpha \circ \alpha^{-1}(S_\phi)$. Since we are over-approximating the system, i. e., $S \subseteq \alpha \circ \alpha^{-1}(S)$, we have 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: $\alpha(S^{\rightarrow}) \subseteq \alpha(S_\phi) \implies S^{\rightarrow} \subseteq S_\phi$. In general, the vice-versa does not hold, which gives rise to counterexample guided abstraction refinement (CEGAR) techniques [6].

4.1 Predicate Abstraction with Arcade.PLC

We extended the tool ARCADE.PLC. Its architecture consists of two main components: *model checker* and *simulator*. The model checker has the task of verifying $\forall$CTL properties on explicit Kripke models. The predicate abstraction allows us to represent a program $\langle S, I, R \rangle$ as a Boolean over-approximation in terms of a Kripke Model $\langle \hat{S}, \hat{I}, \hat{R}, \hat{AP}, \hat{L} \rangle$ where

- $\hat{S}$ is the set of states s. t. $\hat{S} \triangleq \{\alpha(s) \mid s \in S\}$,
- $\hat{I} \subseteq \hat{S}$ is the set of initial states s. t. $\hat{I} \triangleq \{\alpha(s) \mid s \in I\}$,

– $\hat{R} \subseteq \hat{S} \times \hat{S}$ is the transition relation s. t. $\hat{R} \triangleq \{\langle \alpha(s), \alpha(s') \rangle \mid \langle s, s' \rangle \in R\}$,
– $\hat{AP}$ is the set of atomic propositions s. t. $\hat{AP} \subseteq B$ and
– $\hat{L} \colon \hat{S} \to 2^{\hat{AP}}$ is the labeling function s. t. $\hat{L}(\langle \ell, c \rangle) \triangleq \{b \in \hat{AP} \mid c \models b\}$.

The simulator has the task of generating such Kripke model on-the-fly by providing two main primitives: *precondition* and *strongest postcondition*. The precondition $p \subseteq \hat{S}$ corresponds to the set of abstract initial states $\hat{I}$ while the strongest postcondition $sp \colon \hat{S} \to 2^{\hat{S}}$ corresponds to the set of successors of an abstract state under the abstract transition relation $\hat{R}$. Both $\hat{I}$ and $\hat{R}$ are defined as application of the abstraction function over sets. Given a fixed location, this can 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 precondition is defined as the set of abstract states $\langle \ell_a, c \rangle$ at start location which minterms abstract the start condition:

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

The strongest postcondition of an abstract state $\langle \ell, c_1 \rangle$ is 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 respectively:

$$sp(\langle \ell, c_1 \rangle) \triangleq$$
$$\{\langle \ell', c_2 \rangle \mid \langle \ell, \varphi, \ell' \rangle \in G \text{ and } c_1 \wedge c_2' \wedge \varphi \wedge \textstyle\bigwedge_{i=1}^{n}(b_i \leftrightarrow \pi_i \wedge b_i' \leftrightarrow \pi_i') \text{ is } \mathcal{T}\text{-SAT}\}.$$

Computing them reduces to an *AllSAT* problem over a set of important variables [15] that are B for the precondition and B' for the strongest postcondition. Considering the second, in our implementation we iterate among outgoing transitions $\langle \ell, \varphi, \ell' \rangle \in G$ explicitly. For each location l' we query the Z3 SMT solver [8] for models of the formula. From the model we extract a minterm c_2' with which we instantiate an abstract state $\langle l', c_2 \rangle$. After that we conjoin the blocking clause $\neg c_2'$ to the formula and we repeat until unsatisfiable. When iterating among different outgoing transitions directed to the same location l' blocking clauses are maintained in order to avoid double occurrences. The result is memoized.

Example 3. To verify (1) from Sect. 2 we start with $P = \{\pi_1 = (\texttt{out} < 100)\}$ and the property we want to verify thus becomes $AG\, b_1$ (ignoring the exit point for the presentation). Since we do not have any restriction on `var`, we get a counterexample where at the end $\texttt{var} \geq 100$ and is assigned to `out`. This gives rise to the 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 `var`, hence we deduce $\pi_{\text{skip}} = (\texttt{in}_0 < 100)$ (we skip this predicate to make our presentation more accessible). Finally, we discover that the previous statement can only be executed if $\pi_3 = (\texttt{in}_0 + \texttt{in}_1 + \texttt{in}_2 < 100)$ is satisfied (from ℓ_3 to ℓ_4), so π_3 is added to P. The abstracted state space is shown in Fig. 3 (where each b_i represents the validity of π_i) and it allows us to verify (1).

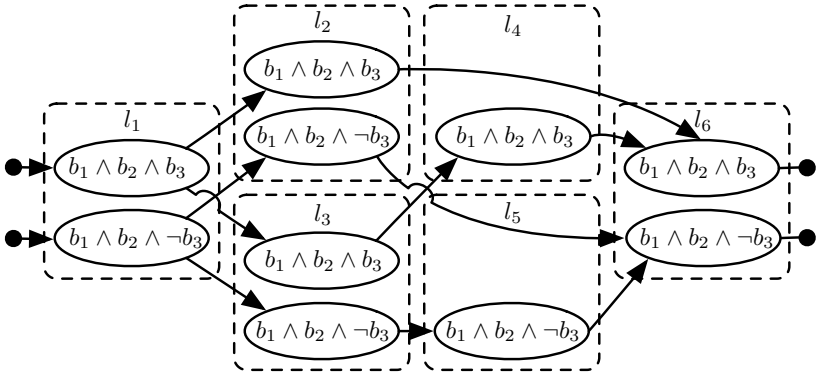

Fig. 3. Predicate abstraction of `Example`. Solid circles on the left and right indicate the scanning phase and are connected by transitions.

4.2 Scoping of Predicates

Thus far, we evaluated all predicates in every location, which is potentially wasteful: Consider the example again with notation as in Ex. 3 and Fig. 3. 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 through ℓ_2. In this section we will therefore reduce the scope of certain predicates and first define:

Definition 7 (Weak Reachability). *Let $\langle L, \cdot, G\rangle$ a control-flow automaton. The weak reachability relation $\preceq \subseteq L \times L$ is defined as follows:*

$$\ell \preceq \ell'' \text{ iff } \ell = \ell'' \text{ or exists } \langle \ell, \cdot, \ell'\rangle \in G \text{ s. t. } \ell' \preceq \ell''.$$

Two locations are weakly reachable if there is a path of locations between them. This path does not consider the transition over data variables, hence two weakly reachable locations could be not actually reachable in a real execution. We associate to each predicate π_i a scope $\langle \check{\ell}_i, \hat{\ell}_i\rangle \in L \times L$ and we redefine the abstraction function in such a way that predicates are used only if they are in the given scope:

$$\alpha(\langle \ell, \nu\rangle) \triangleq \langle \ell, c\rangle \text{ s.t. } c \models b_i \text{ iff } (\check{\ell}_i \preceq \ell \preceq \hat{\ell}_i \implies \nu \models_{\mathcal{T}} \pi_i) \text{ for each } 1 \leq i \leq n.$$

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 $\langle \ell_1, \ell_z\rangle$ and $\langle \ell_a, \ell_m\rangle$, respectively. If the sequence passes through the scanning phase more than once, we do not scope.

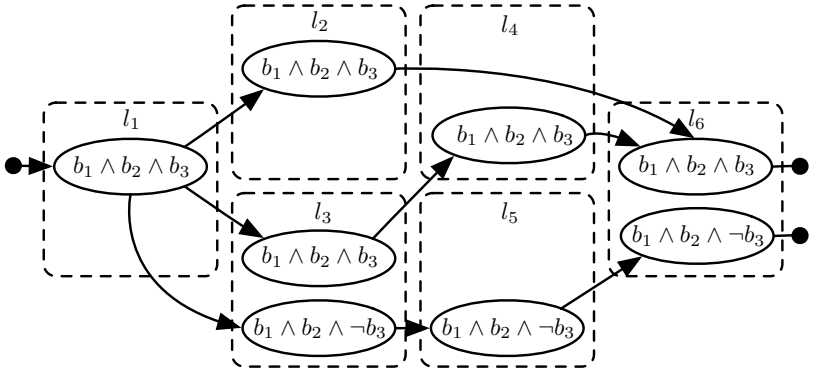

Fig. 4. Predicate abstraction with b_3 scoped to $\langle \ell_3, \ell_6 \rangle$. Solid circles on the left and right indicate the scanning phase and are connected by transitions.

Example 4. In our example we would 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 fixed to 1, hence only state $b_1 \wedge b_2 \wedge b_3$ appears there, further reducing the number of states and transitions. The complete state space is depicted in Fig. 4.

5 Case Study

We implemented our approach in the ARCADE.PLC framework[1] which already offered the necessary parsers and the translation into intermediate code for PLC programs written in *structured text*. To show the effectiveness of our approach beyond the running example, we applied it to various FBs from industrial and academic background and compared it to the existing approach. 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.

Programs For the first experiments, we selected two complex safety-critical FBs proposed by the PLCopen [18] consortium. Since these FBs are defined in terms of state machines without providing an actual implementation we used our own implementation. The first FB `SF_ModeSelector` has 14 inputs, 12 outputs and 5 internal variables and is implemented in 175 lines of structured text. It controls that (up to eight) different modes of operation of a machinery are selected in a consistent way, i. e., that at most one mode is active at a time and only for a short period while switching no mode is selected. It additionally allows the locking of modes. We verified that (1) at most one mode is selected at a time and that (2) exactly one mode is selected if it is locked. Furthermore, we verified the

[1] Aachen Rigorous Code Analysis and Debugging Environment for PLCs: `http://arcade.embedded.rwth-aachen.de`

Table 1. Evaluation with ARCADE.PLC

Program	Formula	Abs.	#loc	#states	#trans.	#P	t_{abs}	t_{total}
`Example`	`out` < 100	—	22	>4k	>40M	n/a	OOM	OOM
`Example`	`out` < 100	PA	22	40	19	4	1 s	1 s
`Example`	`out` < 100	PA+PS	22	10	13	5	1 s	1 s
`SF_ModeSelector`	(1)	—	190	18,759	6.3M	n/a	370 s	370 s
`SF_ModeSelector`	(1)	PA+PS	190	95	142	1	1 s	1 s
`SF_ModeSelector`	(2)	PA	190	>27k	>28k	>40	OOM	OOM
`SF_ModeSelector`	(2)	PA+PS	190	214	291	30	2 s	72 s
`SF_MutingPar`	(3)	PA+PS	377	442	727	1	2 s	2 s
`SF_MutingPar`	(4)	PA+PS	377	>10k	>14k	>100	OOM	OOM
`SafetyFunction`	(5)	PA+PS	442	1,481	2,210	4	4 s	4 s

`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). Finally, we verified a safety applications implemented using PLCopen safety function blocks. The safety application was taken from Soliman and Frey [20] and comprises four FB instances. We were able to verify that the safe stop output is only assumed if it is requested and acknowledged (5).

Evaluation The results are shown in Tab. 1; the columns indicate: the program, the formula checked, the abstraction used ("—" = plain ARCADE.PLC with interval and bit set abstraction, PA = predicate abstraction, PS = 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). Our approach is clearly faster than the old approach and sometimes — as for the example program — even necessary to get a result. The predicate scoping reduces 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). Finally, formula (5) shows how our approach can in principle be used to verify complex safety functions consisting of multiple blocks.

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.

6 Related Work

Our work combines two fields of research, namely the verification of PLC programs by model checking and predicate abstraction of state spaces using decision procedures.

Model Checking of PLC Programs To the best of our knowledge, Moon [16] was the first to work on the formal verification of PLC programs. He translates PLC programs written in a limited subset of *ladder diagram* into the input language of the model checker SVM. This general approach, i. e., rewriting of PLC semantics into the input language of existing model checkers, have been used in numerous paper later on. In 2007, e. g., Pavlovic et al. [17] used NUSMV to verify PLC programs written in a vendor-specific dialect of *instruction list*. This approach, however, required manual insertions of invariants to the model to become feasible. NUSMV was also used by Gourcuff et al. [9] to verify *structured text*, although their approach only supports Boolean variables and a limited subset of control structures. In a second work [10], they made use of abstractions by (a) introducing a notation of observable and unobservable states (induced by the cyclic scanning mode) and (b) abstracting away variables that are overwritten before being used again. Our predicate scoping can be seen as a similar approach although implemented in a much more general way. Schlich et al. [19] then applied direct model checking to PLC programs. This approach, which creates the model using abstract simulation, was later augmented with a CEGAR refinement loop by Biallas et al. [4]. Our work can be seen as a continuation of this line of research.

Predicate Abstraction In their seminal paper, Graf and Saïdi [11] 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 works refined this approach in different directions. Ball et al. [1], e. g., make use of abstraction interpretation [7] 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. [12], on the other hand, introduce a lazy abstraction scheme, which works by using a different precision for different parts of the program which 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.

7 Conclusion

In this paper we advocate a predicate abstraction for PLC programs, which is used to verify $\forall$CTL formulae. We demonstrated that the technique is effective for proving properties in safety-critical programs and operates usually in the order of seconds or minutes. Yet, more powerful techniques seems to be required for the most complicated function blocks we looked at, especially for deriving new predicates.

References

1. Ball, T., Podelski, A., Rajamani, S.: Boolean and cartesian abstraction for model checking c programs. In: Proceedings of TACAS. pp. 268–283. Springer (2001)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handbook of Satisfiability 185, 825–885 (2009)
3. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M., Sebastiani, R.: Software model checking via large-block encoding. In: Formal Methods in Computer-Aided Design, 2009. FMCAD 2009. pp. 25–32. IEEE (2009)
4. Biallas, S., Brauer, J., Kowalewski, S.: Counterexample-guided abstraction refinement for PLCs. In: Proceedings of SSV. pp. 2–9. USENIX Association, Berkeley, CA, USA (2010)
5. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: A verification platform for programmable logic controllers. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 338–341. ACM (2012)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. LNCS, vol. 1855, pp. 154–169. Springer (2000)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
8. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems pp. 337–340 (2008)
9. Gourcuff, V., De Smet, O., Faure, J.M.: Efficient representation for formal verification of PLC programs. In: Proceedings WODES. pp. 182–187 (2006)
10. Gourcuff, V., De Smet, O., Faure, J.M.: Improving large-sized PLC programs verification using abstractions. In: Proceedings of the 17th IFAC World Congress. pp. 5101–5106 (2008)
11. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: CAV. LNCS, vol. 1254, pp. 72–83. Springer (1997)
12. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70. ACM Press (2002)
13. International Electrotechnical Commission: IEC 61131: Programmable Controllers. International Electrotechnical Commission, Geneva, Switzerland (1993)
14. International Electrotechnical Commission: IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems. International Electrotechnical Commission, Geneva, Switzerland (1998)
15. Lahiri, S., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Computer Aided Verification. pp. 424–437. Springer (2006)
16. Moon, I.: Modeling programmable logic controllers for logic verification. IEEE Control Systems Magazine 14(2), 53–59 (1994)
17. Pavlovic, O., Pinger, R., Kollmann, M.: Automated formal verification of PLC programms written in IL. In: VERIFY. pp. 152–163. No. 259 in Workshop Proce., CEUR-WS.org (2007)
18. PLCopen TC5: Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks. PLCopen, Germany (2006)
19. Schlich, B., Brauer, J., Wernerus, J., Kowalewski, S.: Direct model checking of PLC programs in IL. In: Proceedings of DCDS. pp. 28–33 (2009)
20. Soliman, D., Frey, G.: Verification and validation of safety applications based on PLCopen safety function blocks. Control Engineering Practice 19(9), 929–946 (2011), special Section: DCDS'09 — The 2nd IFAC Workshop on Dependable Control of Discrete Systems