Counterexample-Guided Abstraction Refinement for PLCs

Sebastian Biallas, Jörg Brauer and Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
lastname@embedded.rwth-aachen.de

Abstract

This paper presents a method for model checking programs for programmable logic controllers (PLCs) using the counterexample-guided abstraction refinement (CEGAR) approach. The technique is tailored to this specific hardware platform by accounting for the cyclic scanning mode that is symptomatic to PLCs. In particular, the hardware model poses the need for on-the-fly abstraction refinement in order to guarantee a deterministic control flow. It also allows to treat refinement phases triggered by input and global variables differently, leading to a more effective implementation. The effectiveness of this approach is shown in a case study, which highlights the verification process for function blocks that implement a specification provided by the industrial consortium PLCopen.

1 Introduction

Programmable logic controllers (PLCs) are often used to control safety-critical systems, for which formal verification is desirable, if not recommended [17]. Model checking [9] is one particular technique to prove correctness of software written for PLCs. The execution of programs on PLCs follows the so-called cyclic scanning mode, which consists of sensing inputs, processing data, and writing outputs. Each of these steps is executed atomically. A model checker then has to simulate the execution cycle for all possible combinations of input values. Since outputs become visible only at the end of a cycle, internal states within a cycle are not relevant to verification of input-output relations [30].

Programs for PLCs typically depend on several inputs, hence, verification using explicit-state model checking is susceptible to state explosion, since state spaces grow exponentially in the number of inputs. 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.

1.1 Approach

To remedy this problem, we propose to use counterexample-guided abstraction refinement (CEGAR) for ACTL model checking, a technique that has successfully been integrated into several model checkers before [8]. The key idea in CEGAR is to start the verification process on a coarse abstraction of the program semantics. If the specification is satisfied on the abstract semantics, it is also valid in the concrete model. In case the specification is violated, this may be due to abstraction, which manifests itself in a spurious counterexample. In this case, the abstraction is refined in order to obtain a stronger semantics which disallows the behavior that led to the spurious counterexample trace.

Traditional CEGAR techniques, however, allow for nondeterministic control flow, which is not possible for PLCs due to the atomic simulation of a cycle during state space generation. Thus, CEGAR for PLCs requires techniques different from traditional CEGAR approaches as implemented in tools such as SLAM [1] or BLAST [15].

Whereas the refinement step is usually triggered through a spurious counterexample, our method refines the abstraction on-the-fly if atomic propositions cannot be assigned a truth-value during the simulation of a cycle. Further, refinement is triggered to guarantee a deterministic control flow. Based on the scopes of variables, the approach provides two different refinement methods: In 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. This is not so for variables that endure cycles, called *global* variables in this paper[1]. Thus, if global variables trigger the refinement process, the entire state space of the program has to be rebuilt, based on globally refined constraints, so-called *lemmas*.

[1] Note that the term *global* does not refer to the scope of a variable in our description, but to its lifetime.

```
1   VAR_INPUT  input0, input1: BYTE; END_VAR
2   VAR        var0:          BYTE; END_VAR
3   VAR_OUTPUT output0:       BYTE; END_VAR
4         LD     input0
5         ADD    50
6         GT     100
7         JMPC   lbl
8         LD     input1
9         ST     var0
10        RET
11  lbl:  LD     var0
12        ST     output0
13        RET
```

Figure 1: Example program

1.2 Contributions

Overall, we make the following contributions:

1. We describe a symbolic encoding for arithmetic of programs written in Instruction List (IL) that is used to guide the refinement process. Constraint solving over intervals and bit-vectors is used during the refinement itself.

2. We detail a CEGAR-algorithm that is optimized for refinements based on input and global variables, and discuss its implementation in the [MC]SQUARE model checker [29, 30].

3. We show the effectiveness of our method by verifying two function blocks proposed by the industrial consortium PLCopen [31]. Using CEGAR, each of these blocks could be verified on a standard desktop computer, requiring less than 2 minutes per block.

2 Worked Example

We motivate our approach with the example program shown in Fig. 1, which is used throughout the paper. The program has two input variables, a global 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, 50 is added and the result is compared to 100 (lines 4–6). If the result is not greater than 100, the input variable `input1` is copied into the global variable `var0` (lines 8–9). Otherwise, the global variable `var0` is copied into the output variable `output0` (lines 11–12).

To verify this program using naïve methods, a model checker would start by enumerating all possible inputs, creating all successor states. For all successor states this step would be repeated to obtain a state space which can be examined by a model checker. In the example, this approach would create $2^{16} = 65{,}536$ successors for each state, resulting in $2^{32} = 4{,}294{,}967{,}296$ states in total. This is the well-known state explosion, which makes this procedure infeasible for larger programs. To make formal verification possible, abstract states have to be introduced, which represent a (possibly huge) number of concrete states. In this program, for example, it is only relevant whether `input0` lies in the interval $[0, 50]$ or in $[51, 255]$. If we determine that the value lies in either interval, we can decide the conditional jump in line 14 without knowing the concrete value of the accumulator.

To verify programs using this method, the crucial step is to find abstract values such that the behavior of the program is not altered or only altered where it is irrelevant for the validity of the specification. To find abstract values matching these criteria, our approach starts with the most general abstract states representing all possible values. These are then successively refined in order to retain program behavior and the validity of the formula.

In the example, we would assume the abstract value $[0, 255]$ for both inputs and then start simulating the cycle. After loading `input0` and adding 50, the accumulator holds $[50, 305]$. Comparing the latter interval to 100 results in $\{true, false\}$, because the comparison could yield either *true* or *false* depending on the actual concretization. The next operation is a conditional jump, for which the accumulator has to hold a concrete value, since simulating a PLC program simultaneously in two different places is not possible. Thus, the conditional jump poses a restriction on the abstract value in the accumulator.

We call such a restriction a *constraint*. Our key idea is to use this constraint on the abstract value in the accumulator in line 14 to obtain a constraint on the input variables that caused the conditional jump to be ambiguous. The accumulator contains $\{true, false\}$, which is the result of the comparison of $[50, 305]$ with 100. It is therefore sufficient to constrain $[50, 305]$ to be either greater than 100 or less-equal than 100. The interval $[50, 305]$ was the result of the addition of $[0, 255]$ to 50, so we can constrain $[0, 255]$ to be either greater than 50 or less-equal than 50. The interval $[0, 255]$ comes straight from the variable `input0`, so we can now derive that `input0` has to be split into the intervals $[0, 50]$ and $[51, 255]$. This constraint resolving process will later be performed using symbolic information.

Here, the constraint on the accumulator for the conditional jump could be resolved to a constraint on an input variable. By refining the input variable, the problematic values are avoided in subsequent executions after restarting the cycle. Since the values of input variables are assigned independently of previous states, the refinement does not affect already created states. Thus, all constraints on input variables can be resolved by a local restart. The situation is different when it comes to constraints on global variables such as `var0`. Here, splitting

the abstract value might add new program behavior because it was calculated in a previous state. How this is resolved is detailed in Sect. 5.2.

In the next section we briefly introduce the notion of abstract simulation. After that, the constraint solver is detailed which is used for the constraint transformation process. We then formally present the refinement process for input and global variables.

3 Abstract Domains

For abstract interpretation of PLC programs we implemented transfer functions using the interval domain [10] and the bitwise domain [26]. The interval domain combines a set of consecutive integers by storing its upper and lower bounds. While the interval domain is very exact in expressing integer arithmetic as arithmetic on the interval bounds, bit-level operations usually yield inexact intervals containing all values with modified bits. The bitwise domain, on the other hand, represents values as bit-vectors where each bit can either be 0, 1, or *unknown*. Here, bit-level operations are expressed with maximum accuracy using ternary logic, but arithmetic operations usually result in bits becoming unknown. The implementation of transfer functions for this domain has been well-studied in the past [28, 6].

Both abstract domains are combined using the reduced product [10] in a way similar to [28, 6]. This construction ensures taking the more exact description in either of both domains, which is especially important when evaluating truth values of atomic propositions. The combination of both domains allows for precise abstract interpretation of PLC programs as it reflects the most important integer arithmetic and bit-level operations.

4 Constraint Solver

We will first introduce constraints on abstract values, which are then extended to constraints on symbolic expressions. The constraint solver will be used to transform constraints on symbolic expressions into (somewhat equivalent) constraints on variables containing abstract values. In the next section, the constraint solver will be used for our CEGAR approach to select variables for refinement.

4.1 Constraints on Abstract Values

A constraint is a condition f on an abstract value v, denoted $\mathsf{cs}_f(v)$. Such a constraint is fulfilled if the set of concrete values that v represents is *consistent* under the condition defined by f. We introduce the following constraints:

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

A constraint is a restriction on *how many* concrete values an abstract value can represent at most without getting inconsistent. Concrete values are trivially consistent under all constraints (and, vice versa, all constraints can be made consistent by splitting an abstract value into concrete values).

Given a constraint and a variable, we can easily assign abstract values to the variable fulfilling the constraint. Usually, we want these abstract values to cover as many concrete values as possible. This is done using a *splitter*. For a variable v with domain d and a constraint $\mathsf{cs}_f(v)$, the splitter enumerates abstract values $a_1, \ldots, a_n$ such that $\dot{\bigcup}_{i=1}^{n} a_i = d$, a_i is consistent under $\mathsf{cs}_f(d)$ for $1 \leq i \leq n$, and n is minimal. To illustrate, consider a variable v of type `BYTE` and the constraint $\mathsf{cs}_{>100}(v)$. In this case, the splitter would generate the consistent abstract values $[0, 100]$ and $[101, 255]$.

Once we have derived a constraint on a variable, the variable can easily (and efficiently) made consistent by a splitter. The interesting part here is how constraints on arbitrary expressions can be made consistent. In our example, this step was the transformation of the single value constraint to the compare constraint on the variable `input0`. To formalize this process of resolving constraints, we extend the constraints to expressions of abstract values, written $\mathsf{cs}_f(expr)$.

4.2 Constraints on Expressions

To formalize constraints on expressions, we introduce a formal model for representing IL programs. During simulation, the program is written into a symbolic form which explicitly reflects all operations on the accumulator and the variables. For the symbolic representation we use a *static single assignment* (SSA) form [11]. Each left-hand side of an assignment is either the accumulator or a variable. Each right-hand side is either (1) a concrete value (i. e., a constant literal or a variable containing a concrete value), (2) a variable containing an abstract value, (3) a unary operation (complement, negation) on an operand, (4) a data type cast $u_n()$ or $s_n()$ of an operand, where n is the number bits and u (s) signals zero (sign) extension, or (5) an arithmetic or logical operation on two operands.

program	symbolic form	abstract value
`LD input0`	$\mathsf{acc}^{(0)} := {input_0}^{(0)}$	$[0, 255]$
`ADD 50`	$\mathsf{acc}^{(1)} := \mathsf{acc}^{(0)} + 50$	$[50, 305]$
`GT 100`	$\mathsf{acc}^{(2)} := \mathsf{acc}^{(1)} > 100$	$\{true, false\}$
`JMPC label`	$\mathsf{guard}(\mathsf{cs}_{sing}(\mathsf{acc}^{(2)}))$	
`..`		

Figure 2: Program fragment in SSA form

In case an expression results in a concrete value during simulation, 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, which guarantees convergence of our refinement loop.

Figure 2 shows the symbolic and abstract values of the accumulator for the first instructions of the example program (cf. Fig. 1). The transformation of the load, add, and compare instructions into the SSA form is straightforward. Since all calculations with the accumulator are performed over (almost) unbounded integers, we defer all type casting and overflow handling to the store instructions. To decide the conditional jump, we require a concrete value in the accumulator. Therefore, guard nodes 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.

Note that the rewriting is done during the simulation, and hence, all loops are automatically unrolled. Since PLCs have real-time behavior, all cycles have to terminate after a short time, which guarantees bounded size of these symbolic expressions.

The variable constraints from the last section are now extended to expression constraints on SSA expressions. In the next section, we examine how expression constraints such as $\mathsf{cs}_{sing}(\mathsf{acc}^{(2)})$ is transformed.

4.3 Transforming 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 $\mathsf{cs}_{f_1}(e_1) \vdash \mathsf{cs}_{f_2}(e_2)$. To illustrate this, 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:

$$\begin{aligned}
\mathsf{cs}_{sing}(\mathsf{acc}^{(2)}) &\vdash \mathsf{cs}_{sing}(\mathsf{acc}^{(1)} > 100) && (1)\\
&\vdash \mathsf{cs}_{>100}(\mathsf{acc}^{(1)}) && (2)\\
&\vdash \mathsf{cs}_{>100}(\mathsf{acc}^{(0)} + 50) && (3)\\
&\vdash \mathsf{cs}_{>100-50}(\mathsf{acc}^{(0)}) && (4)\\
&\vdash \mathsf{cs}_{>50}({input_0}^{(0)}) && (5)
\end{aligned}$$

In steps (1), (3) and (5), the 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 the addition with the constant. This result allows to fulfill the single-value constraint $\mathsf{cs}_{sing}(\mathsf{acc}^{(2)})$ by refining `input0` into proper abstract values.

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

For an SSA expression $l := e_1$ and a constraint $\mathsf{cs}_f(l)$, we can always apply the transformation $\mathsf{cs}_f(l) \vdash \mathsf{cs}_f(e_1)$. A constraint on a variable terminates the resolving process, whereas constraints on constants do not occur, since constants are trivially consistent under all constraints. The remaining possibilities for SSA right-hand sides are unary operations, binary operations and data type casts. For a unary operation the transformation is defined as follows:

- A complement operation is absorbed by a bit-mask constraint $\mathsf{cs}_{\&\, m}(\neg e_1) \vdash \mathsf{cs}_{\&\, m}(e_1)$.
- A compare constraint on a negation $\mathsf{cs}_{\bowtie c}(-e_1)$ is resolved by $\mathsf{cs}_{\bowtie c}(-e_1) \vdash \mathsf{cs}_{\overline{\bowtie} - c}(e_1)$, where $(\equiv, \overline{\neq}, \overline{<}, \overline{\leq}, \overline{>}, \overline{\geq}) = (=, \neq, \geq, >, \leq, <)$.
- All other constraints on unary operations are resolved as single value constraints $\mathsf{cs}_f(\ominus e_1) \vdash \mathsf{cs}_{sing}(e_1)$.

For a binary operation the transformation is defined as follows:

- A constraint on two non-constant expressions is resolved as a single value constraint on one expression $\mathsf{cs}_f(e_1 \odot e_2) \vdash \mathsf{cs}_{sing}(e_1)$. This other expression is then resolved in the next refinement step.
- For all compare operations $\bowtie$ we resolve $\mathsf{cs}_{sing}(e_1 \bowtie c) \vdash \mathsf{cs}_{\bowtie c}(e_1)$.
- Addition and subtraction in compare constraints are resolved by the translations $\mathsf{cs}_{\bowtie c_1}(e_1 + c_2) \vdash \mathsf{cs}_{\bowtie (c_1 - c_2)}(e_1)$, $\mathsf{cs}_{\bowtie c_1}(e_1 - c_2) \vdash \mathsf{cs}_{\bowtie (c_1 + c_2)}(e_1)$, and $\mathsf{cs}_{\bowtie c_1}(c_2 - e_1) \vdash \mathsf{cs}_{\bowtie (c_1 - c_2)}(-e_1)$.
- Bitwise operation are resolved using the bit mask constraint not presented here.
- All other constraints on binary operations are resolved as single-value constraints on $\mathsf{cs}_f(e_1 \odot c) \vdash \mathsf{cs}_{sing}(e_1)$.

For data type casts the transformation is defined as constraints on the data type bounds. This is not presented in this paper.

Since each 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. This linear complexity is important for our algorithms, because the constraint solver is called for all necessary refinements, as detailed in the next section.

5 Refinements

First, we introduce the formal model that we use to verify PLC programs. A *state* consists of input valuations, output valuations, and global variables of the PLC program at the end of a cycle. In particular, a state specifies the relation between inputs and outputs (and additionally, global variables). While the symbolic SSA representation is used during the simulation of the PLC cycle, we only store the abstract values when saving the actual PLCs states. This means we lose the symbolic descriptions of dependencies between variables, but gain a compact representation of the states. Storing only abstract values also prevents unbounded growth of the symbolic representation, and thus, guarantees convergence of the verification process.

For each state we determine its successors states. The successors of a state are the states reachable by assuming new input values and simulating one cycle. Starting from the initial state of the PLC we obtain a Kripke structure by generating all reachable states [30]. Note that intermediate states during simulation of a cycle are not stored, and thus, do not directly affect the verification results. This allows the program to temporarily take forbidden states that do not influence the outside world. We exploit this unique behavior to develop two different refinement methods.

As we have seen before, the enumeration of all input values easily yields an explosion of the number of states. To tackle this problem, a state consists of a set of abstract values representing a set of concrete values in variables. This way, macro states are created which combine a number of concrete states and so reduce the overall size of the state space.

Our approach for finding this abstraction differs from existing techniques in that it is not solely based on analyzing counterexamples. We especially do not want to introduce nondeterministic control flow while simulating a cycle, which is necessary to prevent the visibility of intermediate states. Instead we track abstract values to the source that generated their value, which is at least one nondeterministic variable — usually an input variable. Splitting the contents of such a variables into smaller abstract values creates separate states, and thus, eliminates the problematic cases.

For the refinement, our method combines two different approaches, depending on whether a local or a global variable has to be refined. Both kinds of refinement are initiated by inconsistent constraints. During simulation, the following situations might occur that require the refinement of an abstract value, and thus, introduce new constraints:

- As we have seen, the control flow has to be deterministic while simulating a cycle. Hence, all conditional instructions (`JMPC`, `CALC`, `RETC`) demand a concrete value in the accumulator.
- Some hardware function blocks (such as timers) require concrete input values for their operation.
- After simulating a cycle, the truth valuations of atomic propositions are determined. The values of the atomic propositions have to be consistent, so they are guarded with appropriate constraints.
- Converting abstract integers into non-integer types (such as floats, strings, etc.) is guarded by single value constraints.

In the first step we will explain how we implemented the refinement of local variables.

5.1 Refinement of Local Variables

Throughout this section, we will assume that all constraints can be fulfilled by refining local variables only (i. e., the refinement algorithm does not have to refine values stored in predecessor states). This is achieved by allowing only concrete values in global variables at the start and the end of each cycle, which is equivalent to adding guards with a single-value constraint to all global variables at the end of a cycle.

Since we do not allow abstract values in the state space in this first step, we do not add additional behavior to the program, and hence, will not find spurious counterexamples. We will see that refinement of local variables is a powerful abstraction of the state space due to the huge number of hidden or combined input values. Our refinement algorithm implements the classical the refinement loop initially described by Kurshan [20], which is embedded into the generation of successor states. It performs the following steps:

1. We store the splitters used for the refinements on a stack. In the first step, a splitter pushed onto the stack that assigns the most broad abstract value (bottom 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.

3. A cycle of the PLC is simulated. 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 and 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.

Using a stack for the splitters ensures that all new refinements are based on and only applied to current input values. This means that the efficiency of this approach is highly dependent on the order in which variables are refined. Since variables are typically referenced in the order of their importance for the control flow in real-world programs, the refinements picked by of our approach are usually quite good. In the next section this method is extended to global variables.

5.2 Refinement of Global Variables

We will now relax the restriction of refining input variables only and allow storing of abstract values in global variables. Since global variables might contain abstract values calculated in previous states, refinement of global variables can create new behavior, i. e., transitions which are not possible in the concrete model. Take, e.g., two variables which contain abstract values, but their concrete value is always 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.

Refining such variables in the abstract model might result in different values, because the symbolic information between the dependency of these variables is lost. But, if an ACTL formula is valid in such an abstract model with added behavior, it is also valid in the concrete model [8]. Otherwise, the formula is violated and we obtain a counterexample. If we had to refine global variables, new behavior was added to the abstract model and counterexamples are not necessarily possible in the concrete model. Such counterexamples are called *spurious*. To verify that a counterexample is not spurious, we rebuild the state space based on a refined semantics.

This approach is depicted in Fig. 3. The first row shows the first iteration of the state space for verifying the formula AG `output0` < 25 (irrelevant states are omitted). In the last state, the truth value of `output0` < 50 is not consistent because `output0` lies in the interval $[0, 255]$, so the state has to be refined accordingly. This is performed using the constraint solver, which finds the constraint $\mathsf{cs}_{>25}(\texttt{var0})$, because 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 $\mathsf{cs}_{>25}(\texttt{var0})$ as a so-called *lemma* for further refinement.

The state space, where the state was split to make the atomic propositions consistent, is shown in the second row. Since `output0` > 25 in the lower state, we have a possible counterexample trace here. However, due to the over-approximation, we have to verify that this counterexample is also valid under concrete semantics.

To achieve this, we rebuild the state space while obeying all lemmas we found, thus avoiding the addition of new behavior to the state space. Therefore, new guards for all variables are added to the end of the program according to their lemmas. The idea is that at the end of the simulation of each cycle — before the state is finally stored — we still have the symbolic information for global variables written in this cycle. Via this guards, the constraint solver either obtains a crucial refinement of an input variable, thus resolving the over-approximation in this state, or it obtains a new lemma, which might be needed in a further refinement/rebuild step.

The final result of the state space obeying $\mathsf{cs}_{>25}(\texttt{var0})$ is shown in the third row of Fig. 3. Here, all additional behavior was removed and we can deduce that the counterexample trace is a real counterexample for the formula AG `output0` < 25.

6 Case Studies

We have implemented the techniques described in this paper in the model checker [MC]SQUARE [29, 30]. To show the effectiveness of CEGAR when applied to the specific task of PLC verification, we checked several functional and non-functional requirements on a number of function blocks proposed by the PLCopen consortium [25]. These implementations were kindly provided by Soliman and Frey [31].

This section highlights the results for two specific blocks that implement *emergency stop* (199 instructions) and *guard locking* (306 instructions), two highly safety-critical tasks. These functions blocks were implemented as state machines according to the PLCopen specification using Boolean variables S_i to indicate their current states.

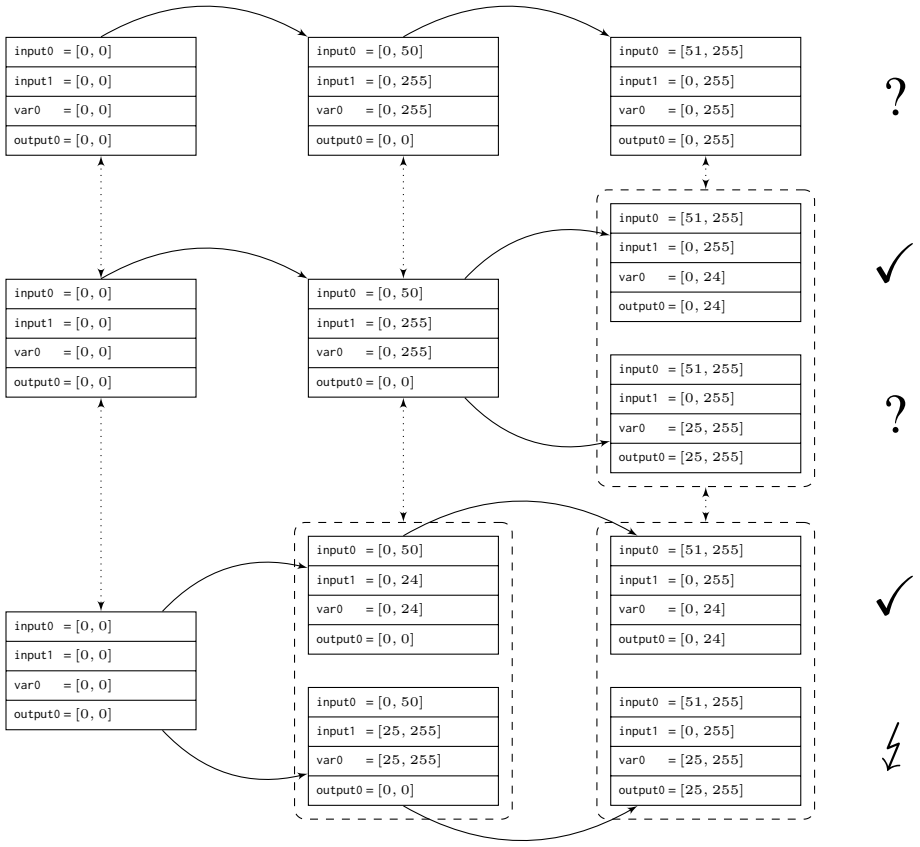

Figure 3: Refinement of state space for verifying AG `output0` < 25

In the implementations, states could be instable or skipped in certain transitions. A typical formula to verify correct transitions from state S_1 is $AG\ (S_1 \implies AX(S_1 \mid S_2))$. Our case study, however, focuses on the effectiveness on generating the state space for different function blocks and not on the actual verification process.

Verifying *emergency stop*, a function block that depends on five Boolean input variables, is easily possibly with [MC]SQUARE without CEGAR. However, even for this program CEGAR proves to be effective, reducing the number of stored states from 134 to 44. Enabling global refinement does not affect the size of the state space at all. For the guard locking implementation, the number of created states was reduced from 199,724,033 to 3,155,467 using local refinement. Further, the runtime was reduced from approximately 100 minutes to 326 seconds. By also enabling global refinement, the runtime was further reduced to 99 seconds, eventually storing only 75,203 states, which highlights the effectiveness of this technique. Using CEGAR, the verification process can be run on a standard desktop computer, which was not possible for larger programs due to the memory requirements.

We have observed comparable reductions for other function block implementations. Space constraints, however, prevent us from presenting these results here.

7 Related Work

Our approach is related to techniques from three fields of research, namely abstraction and refinement in model checking, the verification of software for PLCs, and abstract interpretation. These relations are discussed in the remainder of this section.

Abstract Interpretation

The methodologies used to represent an abstract program semantics date back until the early days of abstract interpretation [10]. In particular, intervals were the first numerical abstract domain used in program analysis [10]. However, it took several decades until it was observed that combining bit- and word-length intervals using the reduced product operator allows to accurately reason about bit-manipulating programs [6, 27, 28]. In contrast to our work, these approaches do not apply any refinement to abstract descriptions.

Program	# Inputs	Options used	States stored	States created	Size [MB]	Time [s]
EmergencyStop	5	None	134	4,289	16	0.56
		Only Inputs	44	307	15	0.20
		All Variables	44	324	15	0.25
GuardLocking	8	None	780,172	199,724,033	1,704	5,633
		Only Inputs	132,242	3,155,467	351	326
		All Variables	75,203	1,098,220	163	99

Figure 4: Results for verifying PLCopen function blocks

Abstraction and Refinement

The idea of refining abstract representions of states in a loop based on encountered over-approximations was first formulated by Kurshan [20]. His observation led to the development of techniques for automated predicate abstraction [13] and the well-known CEGAR-loop [8]. These techniques have found wide application in model checking in different contexts. For instance, Ball et al. [1] apply predicate abstraction and automated abstraction refinement to C code translated into Boolean programs [2]. In contrast, Henzinger et al. [15] 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 their method. Furthermore, the abstraction-refinement scheme has found its way into all areas of model checking, also including bounded model checking [4] using interpolants [12]. Compared to our method, the main difference of existing techniques is that they operate on a general purpose abstraction of the program, whereas our method exploits knowledge about the underlying hardware platform.

PLC Verification

Several attempts have been made in the past to apply model checking to software for PLCs. The first approach goes back to Moon [23], who translated programs given as Ladder Diagrams into the input language of SMV. This approach, however, only supports are very limited subset of Ladder Diagrams (namely, Boolean functions) and does not apply any abstraction, which leads to state explosion for small problems already. Later, Canet et al. [7] verified programs written in IL using NUSMV. The drawback of their method is that they only support a subset of IL and do not account for the cyclic scanning mode. A different approach was followed by Mertke and Frey [21], who translated IL programs into Petri nets, also not supporting the complete IL instruction set.

Huuck [16] used CADENCE SMV to verify PLC programs written as Sequential Function Charts (SFCs). Since parts of the defined SFC constructs have an ambiguous semantics, they only support a well-defined subset of the input language, which is described in [3]. In 2007, Pavlovic et al. [24] described an approach to translate PLC programs in Statement List — a vendor-specific language similar to IL — into the input language of NUSMV. Their approach, however, is not applicable to programs with several inputs without manual intervention. On the other hand, Süflow and Drechsler [32] applied equivalence checking using SAT to the task of PLC verification. Schlich et al. [30] introduced the concept of abstract simulation for PLC verification. This approach, which to a certain degree forms the basis for our work, performs abstraction without refinement, and thus, often leads to spurious warnings. None of the existing techniques, however, embodies an abstraction-refinement loop.

8 Discussion & Future Work

This paper advocates applying a CEGAR approach to model checking of software for PLCs, which integrates the peculiarities of the cyclic scanning mode w. r. t. global and input variables. Whereas the execution of a cycle heavily depends on combinations of inputs, which — in case of concrete instantiations — easily leads to state explosion, a suitable abstraction is automatically derived and refined in our approach. A unique feature of our method is the triggering of refinements in case that nondeterministic control flow is encountered, which can then trigger a global refinement process. This step is required due to the use of a hardware simulator for state space building, in contrast to verification efforts based on, say, Boolean programs [2]. Our CEGAR method proves to be highly effective as it significantly reduces runtimes required for model checking, often by orders of magnitudes.

Clearly, this work calls for further investigations of constraint-solving approaches, and the solver currently in use could be replaced with a back-end based on SAT or SMT [19]. Another possible direction for future research is studying an optimized global refined process, where the state space is not completely rebuilt. Here, existing work on lazy abstraction [15] could serve as a starting point. Further, it is obvious that the domain construction for bit- and word-level intervals is a direct consequence of the early works of Cousot and Cousot [10]. We believe that the effectiveness of this technique can be further improved

by integrating (weakly) relational numeric domains into the refinement process. Octagons [22] or (bitwise) linear congruences [5, 14, 18] are possible choices for suitable relational domains. Such a combination of well-studied domains and the refinement process, however, needs to be examined in detail.

Acknowledgment

The work of Sebastian Biallas was supported by the DFG. The work of Jörg Brauer and Stefan Kowalewski was, in part, supported by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89. We thank Bastian Schlich for sharing his thoughts on the ideas described in this paper, and the anonymous referees for their helpful comments.

References

[1] BALL, T., COOK, B., DAS, S., AND RAJAMANI, S. K. Refining approximations in software predicate abstraction. In *TACAS* (2004), vol. 2988 of *LNCS*, Springer, pp. 388–403.

[2] BALL, T., AND RAJAMANI, S. K. Bebop: A symbolic model checker for boolean programs. In *SPIN* (2000), vol. 1885 of *LNCS*, Springer, pp. 113–130.

[3] BAUER, N., AND HUUCK, R. A parameterized semantics for sequential function charts. In *Semantic Foundations Engineering Design Languages (SFEDL 2002)* (2002), pp. 69–83.

[4] BIERE, A., CIMATTI, A., CLARKE, E. M., STRICHMAN, O., AND ZHU, Y. Bounded model checking. *Advances in Computers 58* (2003), 118–149.

[5] BRAUER, J., KING, A., AND KOWALEWSKI, S. Range analysis of microcontroller code using bit-level congruences. In *FMICS* (2010), vol. 6371 of *LNCS*, Springer, pp. 82–98.

[6] BRAUER, J., NOLL, T., AND SCHLICH, B. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *SCOPES 2010* (2010), ACM.

[7] CANET, G., COUFFIN, S., LESAGE, J.-J., PETIT, A., AND SCHNOEBELEN, P. Towards the automatic verification of PLC programs written in Instruction List. In *SMC* (2000), vol. 4, IEEE, pp. 2449–2454.

[8] CLARKE, E. M., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement. In *CAV* (2000), vol. 1855 of *LNCS*, Springer, pp. 154–169.

[9] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, 1999.

[10] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977), ACM, pp. 238–252.

[11] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Effciently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* (1991), 451–590.

[12] D'SILVA, V., PURANDARE, M., AND KROENING, D. Approximation refinement for interpolation-based model checking. In *VMCAI* (2008), vol. 4905 of *LNCS*, Springer, pp. 68–82.

[13] GIACOBAZZI, R., AND SCOZZARI, F. Intuitionistic implication in abstract interpretation. In *PLILP* (1997), vol. 1292 of *LNCS*, Springer, pp. 175–189.

[14] GRANGER, P. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT 1991* (1991), vol. 493 of *LNCS*, Springer, pp. 169–192.

[15] HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. In *POPL* (2002), ACM Press, pp. 58–70.

[16] HUUCK, R. *Software Verification for Programmable Logic Controllers*. Dissertation, University of Kiel, Germany, April 2003.

[17] INTERNATIONAL ELECTROTECHNICAL COMMISSION. *IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems*. International Electrotechnical Commission, Geneva, Switzerland, 1998.

[18] KING, A., AND SØNDERGAARD, H. Automatic abstraction for congruences. In *VMCAI* (2010), vol. 5944 of *LNCS*, Springer, pp. 281–293.

[19] KROENING, D., AND STRICHMAN, O. *Decision Procedures*. Springer, 2008.

[20] KURSHAN, R. P. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA, 1994.

[21] MERTKE, T., AND FREY, G. Formal verification of plc-programs generated from signal interpreted petri nets. In *SMC* (2001), vol. 4, IEEE, pp. 2700–2705.

[22] MINÉ, A. The octagon abstract domain. *Higher-Order and Symbolic Computation 19*, 1 (2006), 31–100.

[23] MOON, I. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine 14*, 2 (1994), 53–59.

[24] PAVLOVIC, O., PINGER, R., AND KOLLMANN, M. Automated formal verification of plc programms written in IL. In *VERIFY* (2007), no. 259 in Workshop Proce., CEUR-WS.org, pp. 152–163.

[25] PLCOPEN TC5. *Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks*. PLCopen, Germany, 2006.

[26] RAZDAN, R., AND SMITH, M. D. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO* (1994), ACM Press, pp. 172–180.

[27] REGEHR, J., AND DUONGSAA, U. Deriving abstract transfer functions for analyzing embedded software. In *LCTES* (2006), ACM, pp. 34–43.

[28] REGEHR, J., AND REID, A. HOIST: A system for automatically deriving static analyzers for embedded systems. In *ASPLOS* (2004), ACM, pp. 133–143.

[29] SCHLICH, B. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Germany, June 2008.

[30] SCHLICH, B., BRAUER, J., WERNERUS, J., AND KOWALEWSKI, S. Direct model checking of PLC programs in IL. In *DCDS* (2009). To appear.

[31] SOLIMAN, D., AND FREY, G. Verification and validation of safety applications based on PLCopen Safety Function Blocks using Timed Automata in UPPAAL. In *DCDS* (2009). To appear.

[32] SÜLFLOW, A., AND DRECHSLER, R. Verification of plc programs using formal proof techniques. In *FORMS/FORMAT* (2008), L'Harmattan, pp. 43–50.