Abstract. This paper presents a technique for verifying a PLC program’s compliance with respect to the automaton-based specifications used by the PLCopen. While related approaches only support a subset of expressible PLCopen automata and struggle with the state explosion problem, our technique both enables the logical characterisation of any PLCopen automaton and facilitates the verification of programs previously entailing exhaustive state space exploration and time outs. To enable fully symbolic reasoning we utilise the Property Directed Reachability capabilities of the Z3 SMT solver.

1. Introduction

Programmable Logic Controllers (PLCs) are devices especially tailored to the domain of industrial automation. They are robust, standardised and easy to deploy for repeated execution of the same task. Accordingly, they adhere to a cyclic execution mode, i.e. read inputs (typically connected to sensors), execute the main program, write outputs (typically connected to actuators) and start all over again. Programs are usually written in one of the languages defined in the IEC 61131-3 standard. Using the concept of function blocks, consisting of an interface definition and instructions, programs can be modularised – similar to classes in other languages, however, with a single member function. In particular, a program is a function block and may, in turn, call other function blocks.

Despite the safety-critical environments PLCs often operate in, safe behaviour is rarely formalised due to engineers’ unfamiliarity with logics commonly used for specification, e.g. CTL. Thus, with state-based thinking of a program’s behaviour being prominent in the domain of PLC programming, the PLCopen organisation utilises automata for the specification of safe behaviour. Unfortunately the PLCopen does not provide formal semantics for these automata but resorts to illustrating them with examples accompanied by textual descriptions. While the notion of Safety Automata [8] suffices to formalise the most basic PLCopen automata it needs extension to appropriately model the safe behaviour of most function blocks of the PLCopen safety library [11].

Unlike related approaches, we avoid unnecessary exploration of the state space, when possible, by reasoning about the safety of an implementation w.r.t. an automaton’s state locally. To this end, we automatically build a logical characterisation of a single PLC cycle. Utilising the Z3 SMT solver [5], this symbolic representation allows checking whether conditions holding at the beginning of a cycle may result in an unsafe outcome.
Program Structure and Semantics  To get a better idea of PLC program structure and semantics consider Listing 1a, illustrating a main function block’s interface definition and its first instructions. The program makes use of three Boolean inputs, which receive values from sensors connected to the PLC at the beginning of an execution cycle. Furthermore, it contains two local standard function block instances. In particular, R_TRIGatQuery is an instance of the R_TRIG function block type (cf. Listing 1b) which implements rising edge detection. During the execution of a cycle the whole main program, starting in line 15, is executed once. Afterwards, the values at the outputs – in our case only DiagCode – are output to connected actuators and the next cycle begins, where all but the input variables keep their values. Note that calling a function block, as in line 15, does not return a value but merely executes the callee’s instructions. However, a function block can access the members of its local function blocks to acquire the results (cf. line 16).

Contribution and Outline  A survey of related work, outlining the main differences to our approach and enhancements provided by our work, are discussed in Section 2.

Sections 3 and 4 constitute the main contributions of our work. Section 3 covers PLCopen automata, the notion of Safety Automata and Specification Automata, which extend the former. Here we analyse why previous work does not capture the semantics of PLCopen automata properly and propose more suitable semantics to the extension than given in related work. In Section 4 we present our procedure for checking the compliance of a program with respect to a Specification Automaton. We elaborate on the employed encoding and point out opportunities for optimisation. Furthermore, in Section 5, we evaluate both the performance of our approach in contrast to related work and the impact of our optimisations. Finally, we draw concluding remarks in Section 6.

Listing 1: Potential implementation of the running example (cf. Figure 1a) in the Structured Text language
2. Related Work

Current approaches for verification of PLC software can be coarsely divided into ones which check models of the software and others which analyse the actual implementations. For analysis, the former usually transform graphical representations of the software into well-known automata classes, e.g. Sequential Function Charts to Hybrid Automata [10] or Function Block Diagrams and UML to Timed Automata [14, 15, 12]. However, actual implementations of the modelled behaviour may still exhibit unsafe behaviour since machine-integers and overflows must be taken into account and time becomes discrete. Other approaches address this issue by operating on the actual implementation via abstract interpretation [2] or translation into input for bit-precise model checkers [4]. Our technique falls into the latter category since we process the actual implementation and characterise its semantics logically.

To alleviate the hurdle of specifying safe behaviour the PLCopen organisation introduced an informal automata-based formalism. Most work regarding analysis of such automata and their implementations was done by Frey et al., coining the term Safety Automata by giving the first formal definition of basic PLCopen automata [8] where transition conditions can only refer to input variables. In [14], the authors suggest the transformation of both Safety Automata and Function Block Diagrams to Timed Automata. However, their latest joint work with Biallas et al. proposes abstract interpretation of the actual implementation and checking the intersection of reachable and safe behaviour [3]. To enable the analysis of non-basic PLCopen automata, called Specification Automata in [3], they lift the semantics of [14] to state variables. In contrast to this, our technique avoids exploration of the whole state space when possible by reasoning locally and uses a symbolic representation of the program to enable handling of previously problematic semantics, e.g. relational constraints or complex bit level operations. Furthermore, we propose an alternative extension of the basic semantics, for the first time properly capturing the semantics of PLCopen automata.

3. Specification Automata

The PLCopen organisation proposes a set of safety function blocks for common tasks in automation and uses automata to specify the expected behaviour. Such a PLCopen automaton consists of states, labelled with names and invariants, and transitions which have priorities and guards. Intuitively, the semantics of a PLCopen automaton is similar to that of a Moore automaton. When entering or staying in a state, the PLC must satisfy its invariant. With a PLC’s intermediate state not being observable, a transition in the automaton corresponds to a whole PLC cycle and must be taken if its guard is satisfied immediately after reading the inputs. Given the case that several guards are satisfied the transition with the higher priority (lower number) has to be taken. Figure 1a depicts a simple PLCopen automaton which we use as a running example in this paper.

Implementations of state machine based behaviour usually need a variable to keep track of the current state – called DiagCode by the PLCopen. Thus, while invariants are actually propositional formulae over a program’s state variables, i.e. local or output variables, the PLCopen omits the DiagCode variable when clear from context, e.g. the invariant 8000 actually denotes DiagCode=0x8000. In contrast to invariants, guards are propositional formulae over all variables. For example, in Figure 1a, the transition from the wait to the idle state is guarded with answered. This guard is satisfied if the corresponding Boolean input variable is assigned true at the beginning.
of a cycle. Furthermore, guards may also refer to the result of a function block call. For example, the guard \texttt{R\_TRIG \_at \_Query} is satisfied if, at the beginning of a cycle, calling the instance \texttt{R\_TRIG\_at\_Query} of the function block type \texttt{R\_TRIG} (cf. Listing 1a) with the variable \texttt{Query}, will assign \texttt{true} to \texttt{R\_TRIG\_at\_Query.Q}. The PLCopen defines these semantics using timing diagrams [11].

With this in mind, the query-processing modelled by our running example (cf. Figure 1a) specifies the following behaviour. The program starts in the idle-state with \texttt{DiagCode}=0 and stays there until a rising edge is recognised at the input \texttt{Query}. This indicates that some request awaits processing and results in \texttt{DiagCode} being set to the wait-state 0x8000. Here, three outcomes are possible:

1. Setting \texttt{answered} to \texttt{true} indicates successful processing of the request and leads back to the idle-state
2. The time spent waiting for \texttt{answered} to become \texttt{true} may trigger a timeout and lead to the error-state 0xC001
3. If some problem occurs during the processing, setting \texttt{problem} to \texttt{true} leads to the alternative error-state 0xC002

Note that, although the error codes belong to distinct states, they are often illustrated as one due to their shared reset-behaviour.

\textbf{Safety Automata} were introduced in [8] as a formalisation of basic PLCopen automata whose guards may only refer to inputs or the expiration of timers. As part of this formalisation, Safety Automata split combined states like those in Figure 1a into separate states. Furthermore, to avoid ambiguity, they translate both invariants and textual guards into propositional formulae, e.g. the guard "Timer expired" is characterised logically by \texttt{Timer.Q}. Note that referring to timers’ outputs is an exception to the restriction of allowing only input variables in guards. The resulting Safety Automaton comprises the following execution rules:

1. At the beginning of each PLC cycle, the guards of the active state are evaluated in order of their transitions’ priorities
2. Only the transition corresponding to the first satisfied guard fires, i.e. after the cycle the state variables should respect the target state’s invariant

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Comparison of a PLCopen automaton and its formal counterpart}
\end{figure}
3. If no transition can fire, the automaton remains in the active state, i.e. after the cycle the state variables should still respect the source state’s invariant.

A PLC program is considered safe w.r.t. a Safety Automaton if its behaviour adheres to these rules. In particular, if a valuation of the implementation’s variables satisfies a state’s invariant and some guard at the beginning of a PLC cycle, their valuation at the end of this cycle must satisfy the corresponding target’s state invariant.

**Specification Automata** were introduced in [3] as a straight-forward extension of Safety Automata, enabling the use of all variables in the transitions’ guards and allowing under-specification of automata, i.e. implementations may have more behaviour than specified in the automaton. The latter is important in practice since implementations often contain additional behaviour, e.g. logging or vendor specific checks. However, besides allowing the use of all variables in guards, hardly any adaptation of semantics occurred. Inspired by the way in which Safety Automata formalise expiration of timers, the authors characterise calls to function blocks by referring to the respective output variables, e.g. the guard \( R_{TRIG} \) at \( Query \) is simply translated to \( R_{TRIGatQuery}.Q \). However, this handling of both timer expiration and calls does not correspond to the semantics of PLCopen automata. In the following we point out the flaws of previous approaches and propose an adequate handling of calls and timers – yielding proper Specification Automata.

To begin with, reconsider the guard \( R_{TRIG} at Query \) from Figure 1a which checks whether a rising edge is recognised when \( Query \) is passed to the \( R_{TRIG} \) instance. Treating this guard like \( R_{TRIGatQuery}.Q \), as done in related work, merely checks the output-variable of the \( R_{TRIG} \) instance – thereby ignoring the call semantics and referring to an old valuation of \( Q \). Furthermore, note that a timer is a function block, too, and thus only changes its output values when being called. Accordingly, logically characterising "Timer expired" as \( Timer.Q \) does not properly poll the timer’s state but suffers from the same problem as all guards using calls. We, in contrast, do not distinguish between timers and other function blocks but take them into account in a uniform manner.

**Characterising Calls** We observed that in practice guards may contain calls of two different types: the *parameterised* ones, where all parameters are specified, and *unparameterised* ones, where at least some parameter is not specified. Most calls occurring in guards fall into the former category, as does \( R_{TRIG} at Query \) from our example by specifying \( Query \) as parameter. Timer calls, however, are typically missing a parameter. A standard timer function block has two inputs: a Boolean \( IN \) and a time bound \( PT \). While the time bound is typically constant and known, PLCopen automata typically under-specify the use of timers by not stating what is passed to \( IN \) (cf. Figure 1a), thereby only specifying what should happen once timer expiration is detected.

In the case of parametrised calls in guards, the call already specifies for which parameters it is supposed to assign *true* to the output variable and can therefore be characterised logically by the symbolic expression which relates the callee’s input valuation with the output. For example, the output \( Q \) of a \( R_{TRIG} \) is determined by \( CLK \) and \( not M \), as can be seen in Listing 1b. Therefore, when encountering a guard \( R_{TRIGatQuery}(CLK:=Query) \) we actually operate on the expression \( Query \) and \( not M \) instead. While the relation is easy to see for the example, it may be a lot more complex in practice. Therefore, we employ symbolic execution, similar to [13], to compute the relation.
In the case of unparameterised calls the specification does not bind the callee’s inputs to specific variables but asserts the result to be true after the first call. This, in turn, will indirectly affect other variables. For example, when encountering a guard Timer expired, the verification procedure has to back-propagate the assertion that Timer.Q must be true after the first call. Using a symbolical approach, this can be realised elegantly during the encoding.

Figure 1b illustrates our internal representation of the Specification Automaton underlying the running example, where both kinds of calls are explicitly represented. Note that we implement the body of timer function blocks similar to [7], abstracting from the precise value since it is not known how much time any instruction takes.

4. Symbolic Verification

The main focus of our approach is to avoid the costly construction and exploration of the state-space which usually impedes model-checking approaches. A key observation in the context of checking function blocks against Specification Automata is that safety of an implementation w. r. t. a state can often be verified locally. For example, whether the implementation properly reacts to a timer expiration is typically independent of the execution history. In line with this, we verify a program’s compliance w. r. t. to a Specification Automaton by considering each of its states separately and check, assuming that the program is in the state, whether all outgoing transitions can be taken by the program in the specified way – incorporating the transitions’ priorities.

4.1. Concept

An overview of our approach is visualised in Figure 2. Firstly, the program under analysis is translated into an intermediate representation (IR). This has the advantage of the underlying verification procedure being independent from the chosen programming language. This IR is then appropriately encoded for the Property Directed Reachability (PDR) engine of Z3 (cf. Section 4.2). Note that the program itself is encoded just once.

Subsequently, the given Specification Automaton is processed state-wise. In the following, let $S$ denote the set of states. For each state $source \in S$, we collect the outgoing transitions $\Pi_{source} = \{(b_1, target_1), \ldots, (b_n, target_n)\}$, which are pairs of guards and target states, ordered by descending priority. Note that implicit self-loops, for the case that no guard is satisfied, are added to each state with lowest priority. Let those already be contained in each $\Pi_{source}$.
To check whether the implementation complies with the specified safe behaviour for a state \( source \in S \), we perform the following steps:

1. Assert the program’s encoding for a single cycle and the reachability of the cycle-entry state
2. Determine the source state’s invariant \( \varphi_{source} \)
3. Pick the transition \( \langle b_i, target_i \rangle \) with the highest-priority, which has not been considered yet
4. Encode the unsafe behaviour
   \[
   \varphi_{bad} := \varphi_{source} \land (\neg b_1 \land \cdots \land \neg b_{i-1}) \land b_i \land \neg \varphi_{target_i} \land \varphi_{atExit},
   \]
   satisfiable by a single PLC cycle which
   • starts in \( source \),
   • satisfies the \( i \)-th transition’s guard (but none of higher priority),
   • does not end up in \( target_i \),
   and is evaluated at the cycle exit.
5. Call the PDR engine of Z3 to check the reachability of unsafe behaviour \( \varphi_{bad} \) under the given assertions
6. If the result is \( SAT \), retry the check with two additional assertions, closing the PLC cycle and assigning the initial values in the very first state
7. Go to step 3 if there exist transitions not considered yet

Note that referring to \( b_i \) only in \( \varphi_{bad} \) would not suffice to characterise the unsafe behaviour since guards of higher priority might be satisfied, too. Thus, when considering a transition, we conjunct the negated guards of previous transitions, i.e. add \( (\neg b_1 \land \cdots \land \neg b_{i-1}) \land b_i \).

By encoding a single PLC cycle and only asserting the currently considered state’s invariant, the result of a check refers to all cycles satisfying the invariant – over-approximating the reachable state space. Therefore, if a check yields \( UNSAT \), the implementation is proven to not violate the automaton’s safety constraints on the checked transition. However, if a check yields \( SAT \), the resulting variables’ valuation might actually be a spurious counterexample, i.e. unreachable from the program’s initial state. This is why we temporarily add both the assertion which encodes the cyclic behaviour and the one which assigns the initial values, for the duration of a check falling back to non-local but precise symbolic reasoning.

### 4.2. Encoding

Property Directed Reachability (PDR) [6], also known as IC3, is an algorithm for checking the reachability of a set of bad states, given a transition relation and an expression which characterises initial states. In essence, starting in the initial states, the algorithm tries to find an inductive invariant, proving the unreachability of bad states, by computing a fixed-point of the set of unreachable states in an incremental way. To utilise Z3’s PDR-capabilities we logically characterise the implementation’s
semantics in terms of a function $state$, characterising the set of reachable states, and a relation $step$, modelling direct reachability between two states.

In the following, we present the idea behind our encoding by means of the running example (cf. 1b) and the check whether the transition from the wait- to the error1-state can be violated by an implementation.

**Logical Characterisation of Implementation** Let us begin with the encoding of the PLC program itself. Since a program’s state is essentially the current program location $l$ and the valuation of its variables, one might expect it to be sufficient to use $state(l, x)$ to characterise states and the relation $step(l, x, l', x')$ between source- and destination-variables instances to characterise direct reachability – employing $x$ to denote the vector of all of the implementation’s variables. This, however, does not suffice since we must be able to refer to both the valuation at the entry and at the exit of a PLC cycle at the same time. For example, variables occurring in guards refer to the valuation at the entry while those occurring in a target state’s invariant refer to the valuation at the cycle exit. To this end, we extend $state$ with another vector of variables $x_{cpy}$, to keep a copy of the variables’ valuation through the whole cycle. As a result, we end up with using

$$state(l, x, x_{cpy}) \land step(l, x, x_{cpy}, l', x', x'_{cpy})$$

to characterise reachable states and their connection. For reasons of clarity, we give a simplified and intuitive description of how the implementation can be encoded, given a Control Flow Graph (CFG) of the program. Since every edge can be translated separately, let us consider an edge from program location $l_{src}$ to $l_{dst}$ representing an instruction $instr$. The asserted rule, respectively all-quantified Horn formula, would be

$$(\llbracket instr \rrbracket \land x_{cpy} = x'_{cpy}) \Rightarrow step(l_{src}, x, x_{cpy}, l_{dst}, x', x'_{cpy}),$$

where $\llbracket instr \rrbracket$ denotes the logical characterisation of the instruction, e. g. $\llbracket assume(y > z) \rrbracket = y > z \land x = x'$ with both $y$ and $z$ being contained in $x$. Note how $x_{cpy} = x'_{cpy}$ implements the previously mentioned carrying of the entry valuation through the whole cycle.

While such an encoding suffices to grasp the concept, it actually performs rather bad in practice. Therefore, our implementation computes a Cutpoint Graph (CG), where each node is either a loop head or the entry or exit of the CFG, and each edge in the CG corresponds to a loop-free path through the CFG [9]. With this, instead of encoding each edge of the CFG, we encode each edge of the CG, which is equivalent to employing Large Block Encoding [1] for each loop-free segment of the CFG and is significantly more efficient for PDR.

To establish the connection between the function $state$ characterising reachable states and the transition relation $step$, we assert the rule

$$state(l, x, x_{cpy}) \land step(l, x, x_{cpy}, l', x', x'_{cpy}) \Rightarrow state(l', x', x'_{cpy}),$$

corresponding to the transitive closure of $step$. The assertion needed to close the PLC cycle (cf. Section 4.1, Step 6) is straightforward, connecting the cycle exit $l_{exit}$ and entry $l_{entry}$ locations, which are known statically, and binding all but the inputs to their old values:

$$state(l_{exit}, x, x_{cpy}) \land state(l_{entry}, x', x'_{cpy}) \land x' = x'_{cpy} \land \bigwedge_{\text{Inputs}} (v = v' \land v_{cpy} = v'_{cpy})$$

$$\Rightarrow step(l, x, x_{cpy}, l', x', x'_{cpy})$$
Logical Characterisation of Constraints  Now that the program can be logically characterised, let us focus on encoding the constraints mentioned in the algorithm from Section 4.1. Remember that we actually wanted to check whether the transition from the wait- to the error1-state in the running example (cf. 1b) can be violated by an implementation. We begin by asserting the reachability of the program’s first statement (cf. Section 4.1, Step 1) through the rule

\[(x = x_{cpy}) \Rightarrow state(\text{l}\text{entry}, x, x_{cpy}),\]

not constraining the variables’ valuation except for being identical for both \(x\) and \(x_{cpy}\).

Let us now focus on constructing the parts necessary for \(\varphi_{bad}\) and begin with determining the source state’s invariant \(\varphi_{source}\) (cf. Section 4.1, Step 2). To state that the variables’ valuation at the beginning of a PLC cycle must satisfy \(DiagCode = 0x8000\) we let

\[\varphi_{source} = (DiagCode_{cpy} = 0x8000),\]

however, since \(\varphi_{bad}\) will be checked at the exit-location only, we refer to \(DiagCode_{cpy}\) from \(x_{cpy}\) instead of the plain \(DiagCode\) whose value will typically have changed during the cycle. In contrast, the target’s invariant \(\varphi_{target2}\) must refer to the variables from \(x\) to evaluate it over the cycle-exit valuation of the variables, i.e.

\[\varphi_{target2} = (DiagCode = 0xC001).\]

To make sure that the valuation of \(x\) in \(\varphi_{bad}\) is indeed the one at the cycle exit, \(\varphi_{atExit}\) characterises reachable states at the exit location:

\[\varphi_{atExit} = state(\text{l}\text{exit}, x, x_{cpy}).\]

So far, we have taken everything but the transition’s guard into account. However, knowing that there is a transition from the wait- to the idle-state with higher priority we have to exclude the possibility of its guard being satisfied in the first place (cf. Section 4.1, Step 4):

\[\varphi_{bad} := \varphi_{source} \land \lnot b_1 \land b_2 \land \lnot \varphi_{target2} \land \varphi_{atExit},\]

where \(b_1 = answered_{cpy}\). Note that we use \(answered_{cpy} \in x_{cpy}\) to make sure that we refer to the valuation of \(answered\) carried from the beginning of the cycle and not to one which was possibly modified during the execution of the cycle.

Now, it merely remains to encode \(b_2\), i.e. the unparameterised call \(Timer()\). Since calling a function block without parameters in a guard means that the corresponding Q is \textit{true} after the block’s first invocation in the implementation, we must find a way to propagate the valuation of Q to the end of the cycle, s.t. we can refer to it. We realise this by instrumenting such function blocks with the additional variables \(calledOnce\) and \(QAfterFirstCall\) and appending \texttt{if not calledOnce then QAfterFirstCall:=Q; calledOnce:=true} to their body. As a result, the variable \(QAfterFirstCall\) still holds the result of the first call, at the end of a PLC cycle. Accordingly, a call \(Timer()\) occurring in a guard can now be treated like the guard \(Timer.calledOnce\) and \(Timer.QAfterFirstCall\), evaluated using the variables’ valuation at the end of a cycle, i.e. not the one from \(x_{cpy}\). With this in mind, we can now complete \(\varphi_{bad}\) for the considered transition with:

\[b_2 = Timer.calledOnce \land Timer.QAfterFirstCall\]

where \(Timer.calledOnce\) and \(Timer.QAfterFirstCall\) are elements of \(x\).
4.3. Optimisations

As illustrated in Figure 2, our verification procedure is designed in a way which allows checking of an implementation’s safety w. r. t. to an automaton’s state independently. This circumstance gives rise to parallelisation of this procedure. For this purpose, we employ a solver instance for each state and run the procedure from Section 4.1 in concurrent threads.

Although we could go one step further and even check the automaton’s transitions concurrently, we restrain from this as it would hardly affect the utilisation of the CPUs but hinder us from exploiting the incremental nature of the check. With Z3 supporting incremental solving, it is desirable to reuse the same solver object for checks which share assertions. A closer look on our verification procedure reveals that processing the transitions of a state in order of their priority goes hand in hand with an increasing shared prefix of assertions and queries. When checking the $i$-th transition we already assert the program’s encoding and the source state’s invariant. However, instead of making $\neg b_1 \land \cdots \land \neg b_{i-1}$ part of $\phi_{\text{bad}}$ we may as well assert this expression since it will occur in checks of transitions with lower priority, thereby enabling the solver to keep auxiliary information in between checks instead of deducing it over and over again.

5. Evaluation

We evaluated our approach on implementations of 12 function blocks taken from the PLCopen safety library [11] and a more complex automation program we use at our chair. Our results are compared to the abstract interpretation based approach of [3], representing the only currently available technique for checking programs against PLCopen automata. Although they do not model the semantics of PLCopen automata faithfully, a comparison of runtimes lends itself to examine the fundamental feasibility of both approaches. To examine the impact of our optimisations we ran our verification procedure under different configurations, toggling the use of parallel computation ($\text{(np)}$ – (non) parallel) and incremental usage of the solver ($\text{(ni)}$ – (non) incremental). All computations were conducted on a laptop with 16GB RAM and a Core i7-4700MQ processor with 2.4GHz, using the Windows 7 64bit operation system – the maximal memory usage was merely...
600MB (by abstract interpretation) though.

Our measurements of the runtime are listed in Table 1. Runs that took longer than an hour were aborted – denoted by TO. We find that for simple function blocks, where abstract interpretation can explore the whole state space efficiently, both approaches achieve negligible verification times, ranging from 19ms to 501ms. Here the overhead of logically characterising the program is higher than the actual check and the fact that the abstract interpretation based approach is often faster on these blocks is not relevant in practice.

The results achieved for more complex blocks are more revealing. For example, consider the implementations of SF_GuardLocking and SF_GuardMonitoring which have quite similar semantics. The most striking difference is that the former employs an additional R_TRIG instance. However, this increase in complexity seems to account for the exponential growth of the runtime for the approach of [3]. The PDR-based approach, on the other hand, is hardly affected by this and only takes 246ms for verification. Any more complex blocks seem to pose a problem for the abstract interpretation based approach and time out. The proposed approach, however, manages to analyse every function block in a few seconds. As the “Safe” column indicates, our approach managed to verify the correctness of all but two function blocks and found bugs in the others. Verification of a "real" program RTSort, which handles monitoring and re-routing of balls on track, is not possible with [3] either. Yet our approach scales sufficiently to verify its correctness in short time. Note that the different logical characterisations seem not to be significant for the runtime differences since ours is more intricate, due to proper the handling of function block calls.

Table 1 reveals the impact of the optimisations proposed in Section 4.3. Whereas the exploitation of incrementality only yields a 7% speed-up, parallel execution performs 66% faster than the sequential implementation. Combining both optimisations yields an average speed-up of 71%.

As pointed out in Section 3, the PLCopen automaton semantics implemented in [3] differ from ours. Accordingly, the results of our verification procedures differ for some of the non-trivial function blocks. For example, while the SF_SafetyLimitedSpeed function block was verified to be correct by our technique, the abstract interpretation based approach returned an non-realisable counterexample due to the wrong modelling of calls – specifically R_TRIG.

6. Conclusion

In this work we presented an approach which for the first time enables checking the compliance of a PLC program with any PLCopen automaton. To avoid the state space explosion problem, which inhibits prior model-checking approaches in this area, it reasons locally (to an automaton’s state) when possible and utilises the PDR algorithm for symbolic reasoning.

Furthermore, we pointed out flaws of related approaches in the adoption of PLCopen automata semantics and proposed appropriate changes.

Our technique was evaluated on a selection of safety function blocks of the PLCopen and a more complex program from our chair. Experimental results indicate that, while yielding verification times similar to an abstract interpretation based approach for simple programs, our technique scales up to more complex function blocks which are intractable for the former approach.

In future work, we want to add support for more comprehensible counterexamples from the output that is generated by Z3 as the currently output set of predicates is hardly helpful for an engineer. Since in some cases we resort to reasoning globally, we also want to investigate whether this is really necessary – it might suffice to reason over a smaller part of the implementation.
References


